# Introduction

## What Is Algorithm?

Let us consider the problem of preparing an omelet dish. To do that , we follow the steps given below:

1. Get the frying pan.
2. Get the oil.
   1. Do we have oil?
3. If yes, put it in the pan.
4. If no, do we want to buy oil?
5. If yes, then go out and buy.
6. If no, we can terminate.
7. Turn on the stove, etc...

What we are doing is are providing a step-by-step procedure for solving it.

Formal definition of an algorithm: **An algorithm is the step-by-step unambiguous instructions to solve a given problem.**

## What Is Analysis of Algorithms?

Multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort, etc.). Algorithm analysis helps us **determine which algorithm is most efficient in terms of time and memory consumed**.

## How to Compare Algorithms?

Do you think following measures are good to compare algorithms?

* ~~Execution times~~? Not a good measure as they are specific to a particular computer.
* ~~Number of statements executed~~? Not a good measure as it varies with the programming language and the style of the individual programmer.

Ideal solution? Let us assume that we express the running time of a given algorithm as a function of the input size n (f(n)) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc.

## What is Rate of Growth?

The rate at which the **running time increases as a function of input** is called *rate of growth*.

Below is the list of growth rates you will come across in the following chapters:

|  |  |  |
| --- | --- | --- |
| **Time Complexity** | **Name** | **Example** |
| 1 | Constant | Adding an element to the front of a linked list |
| logn | Logarithmic | Finding an element in a sorted array |
| n | Linear | Finding an element in a unsorted array |
| lognlogn |  |  |
| nlogn | Linear Logarithmic | Sorting n items by ‘Divide and Conquer’ |
| n2 | Quadratic | Shortest path between 2 nodes in a graph |
| n3 | Cubic | Matrix Multiplication |
| 2n | Exponential | The Towers of Hanoi problem |

## How Many Types of Analysis?

There are three types of analysis:

1. **Worst case**

* Defines the input for which the algorithm takes **the longest time to complete**.

1. **Best case**

* Defines the input for which the algorithm takes **the fastest time to complete**.

1. **Average case**

* Provides a prediction about the running time of the algorithm.
* Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
* Assumes that the input is random.

Lower Bound <= Average Time <= Upper Bound

## Notation

### Big-O Notation

This notation gives the tight upper bound of the given algorithm and we represent it as f(n) = O(g(n)).

For example, if f(n) = n4 + 100n2 + 10n + 50 is the given algorithm, then g(n) is n4 => O(n4).

**O(1)**

Time complexity of a function (or set of statements) is considered as O(1) if it doesn’t contain loop, recursion and call to any other non-constant time function. For example:

// set of non-recursive and non-loop statements

A loop or recursion that runs a constant number of times is also considered as O(1). For example:

// Here c is a constant

for (int i = 1; i <= c; i++) {

    // some O(1) expressions

}

**O(n)**

Time complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example:

// Here c is a positive integer constant

for (int i = 1; i <= n; i += c) {

    // some O(1) expressions

}

**O(nc)**

Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example – the following sample loops have O(n2) time complexity:

for (int i = 1; i <= n; i += c) {

    for (int j = 1; j <=n; j += c) {

        // some O(1) expressions

    }

}

**O(logn)**

Time complexity of a loop is considered as O(logn) if the loop variables is divided / multiplied by a constant amount. For example:

for (int i = 1; i <= n; i \*= c) {

    // some O(1) expressions

}

**O(loglogn)**

Time complexity of a loop is considered as O(loglogn) if the loop variables is reduced / increased exponentially by a constant amount. For example:

// Here c is a constant greater than 1

for (int i = 2; i <= n; i = pow(i, c)) {

    // some O(1) expressions

}

### Omega-Ω Notation

This notation gives the tight lower bound of the given algorithm and we represent it as f(n) = Ω(g(n)).

For example, if f(n) = 100n2 + 10n + 50 is the given algorithm, then g(n) is n2 => Ω(n2).

### Theta-Θ Notation

This notation decides whether the upper and lower bounds of a given algorithm are the same.

IMPORTANT NOTE:

In the remaining chapters, we generally **focus on the upper bound (O) because knowing the lower bound (Ω) of an algorithm is of no practical importance**, and we use the Θ notation if the upper bound (O) and lower bound (Ω) are the same.

# Visualization Tools

Data Structures and Algorithms Visualization Tool: <https://csvistool.com/>

Tree Drawing and Visualization Tool: <https://tree-visualizer.netlify.app/>

Graph Drawing Tool: <https://csacademy.com/app/graph_editor/>

# Recursion and Backtracking

# Dynamic Programming

# Recursion

<https://www.youtube.com/watch?v=ngCos392W4w&ab_channel=Reducible>

## One Function Call

Example:

#include <iostream>

using namespace std;

void func(int n) {

if (n > 2) {

func(n - 1);

}

cout << n << endl;

}

int main() {

func(5);

return 0;

}

Output:

|  |  |
| --- | --- |
|  | 2  3  4  5 |

## Multiple Function Calls

Example:

#include <iostream>

using namespace std;

void func(int n) {

if (n > 2) {

func(n - 1);

func(n - 2);

func(n - 3);

}

cout << n << endl;

}

int main() {

func(5);

return 0;

}

Output:

|  |  |
| --- | --- |
|  | 2  1  0  3  2  1  4  2  1  0  3  2  5 |

Explanation:

<https://www.youtube.com/watch?v=0nKIr3kAt-k&ab_channel=GinaSprint> (from 4:18 to 12:35)

<https://www.youtube.com/watch?v=B3U6LExgevE&ab_channel=BytebyByte>

# Data Structures

## Array

std::array

[c++ - std::array vs array performance - Stack Overflow](https://stackoverflow.com/questions/30263303/stdarray-vs-array-performance)

## Vector

Vector elements are placed in contiguous memory blocks.

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Time Complexity** | **Space Complexity** | **Note** |
| sort(v.begin(), v.end()) | O(nlog(n)) | O(log n) | Time: Comparison-based sorting algorithm like Quicksort or Mergesort. |
| reverse(v.begin(), v.end()) | O(n) | O(1) | Space: Operates in-place, without requiring additional memory. |
| v.push\_back(val) | O(1) | O(1) | Time: In worst case, when not enough capacity for contiguous blocks, vector needs to reallocate memory a new larger block, so O(n).  Space: In worst case, reallocating memory takes O(n). |
| v.pop\_back(val) | O(1) | O(1) |  |
| v.size() | O(1) | O(1) |  |
| v.clear() | O(n) | O(1) |  |
| v.erase(iterator) | O(n) | O(1) | Time: Needs to shift the remaining elements to fill the gap left by the erased element |
| v.insert(pos, val) | O(n) | O(1) | Time: Needs to shift the existing elements after the insertion point to make room for the new element |
| v.back() |  |  |  |

## Linked List

Great video about how to implement linked list in C:

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjFiudyP6wSXmykrn67Ykqib>

### Definition

### Operations & Complexities

### Applications

### Linked List vs Array

Both arrays and linked list can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.





**Drawbacks of arrays:**

1. The size of the arrays is fixed: We must know the upper limit on the number of elements in advance. Also, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.

2. Inserting a new element to an array is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

For example, suppose we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040, ...].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

3. Deletion is also expensive with arrays until unless some special techniques are used.

For example, to delete 1010 in id[], everything after 1010 has to be moved.

**Linked list provides following two advantages over arrays:**

1. Dynamic size

2. Ease of insertion/deletion

**But linked lists have following drawbacks:**

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So, we cannot do binary search with linked lists.

2. Extra memory space for a pointer is required with each element of the list.

3. Arrays have better cache locality that can make a pretty big difference in performance.

### Why double pointers are used in linked list?

It is clear that both methods (single pointer and double pointer) lead to the **same result**. The only difference is **what will be changed afterward**.

Double pointers are used as **arguments** of function when the function modifies and updates the linked list without needing to return the value (address or data) of the list again.

When using single pointers as arguments of function that modifiers and updates the linked list, we must return the value (address or data) of the list. Or else, the effect won’t be noticed.

Briefly, remember the simple C rule: If you want to **modify local variable of one function inside another function**, pass pointer to that variable. It is called "call by pointers". In this case, the pointer is C’s way of implementing "call by reference" when there is no reference variable.

For example, you want to add a new node before the head (first node) of the list, and hence, the pointer pointing to the first node will be then changed. When you exit this function, you want this change to reflect in the calling function and the following code in the main() function (suppose you call this function in the main()). In this case, you have to use a double pointer. One of them is to indicate that you are passing an address and another is to make the changes available to the calling function (to achieve call by reference).

## Stack

### Definition

Stack is a linear data structure that allows adding and removing elements in a specific order. In particular, every time an element is added, it goes on the top of the stack. The only element that can be removed is the one at the top of the stack. In other words, **the first item added to a stack will be the last item removed from it**. As a result, a stack is said to have "last in first out" behavior (or *LIFO*).

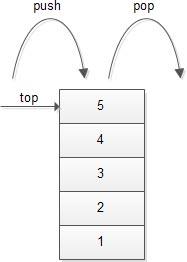
A typical example of using stack is function calling. A function calls another function, which in turn calls a third function; it's important that the third function return back to the second function rather than the first one.

*You might not know!*

The "call stack" is the term used for the list of functions either executing or waiting for other functions to return.

### Operations & Complexities

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Meaning** | **Time Complexity** | **Space Complexity** |
| S.push(val) | Adds an item in a stack. If the stack is full, it is said to be an *Overflow* condition | O(1) | O(1) |
| S.pop() | Removes an item from a stack. If the stack is empty, it is said to be an *Underflow* condition | O(1) | O(1) |
| S.top() or S.peek() | Returns a reference to the top most element of a stack | O(1) | O(1) |
| S.empty() | Returns true if stack is empty, else false | O(1) | O(1) |
| S.full() | Returns true if stack is full, else false | O(1) | O(1) |
| S.size() | Returns the size of a stack | O(1) | O(1) |



### Applications

* [Balancing of symbols](https://www.geeksforgeeks.org/check-for-balanced-parentheses-in-an-expression/)
* [Infix to Postfix](http://quiz.geeksforgeeks.org/stack-set-2-infix-to-postfix/) /Prefix conversion
* Redo-undo features at many places like editors, photoshop
* Forward and backward feature in web browsers
* Used in many algorithms like [Tower of Hanoi,](https://www.geeksforgeeks.org/recursive-functions/)[tree traversals](https://www.geeksforgeeks.org/618/), [stock span problem](https://www.geeksforgeeks.org/the-stock-span-problem/), [histogram problem](https://www.geeksforgeeks.org/largest-rectangular-area-in-a-histogram-set-1/).
* Other applications can be Backtracking, [Knight tour problem](https://www.geeksforgeeks.org/backtracking-set-1-the-knights-tour-problem/), [rat in a maze](https://www.geeksforgeeks.org/backttracking-set-2-rat-in-a-maze/), [N queen problem](https://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/) and [sudoku solver](https://www.geeksforgeeks.org/backtracking-set-7-suduku/)
* In Graph Algorithms like [Topological Sorting](https://www.geeksforgeeks.org/topological-sorting/) and [Strongly Connected Components](https://www.geeksforgeeks.org/strongly-connected-components/)

### Implementation

There are some ways to implement a stack:

* Using array
* Using linked list
* Using vector

**Example 1**: Using array (🡪 static stack)

#include <bits/stdc++.h>

using namespace std;

#define MAX\_SIZE 1000

class Stack

{

private:

    int top;                    // Index of the top item

public:

    int arr[MAX\_SIZE];          // Maximum size of Stack

    Stack() {

        top = -1;

    }

    bool push(int x) {

        if (isFull()) {

            cout << "Stack Overflow";

            return false;

        }

        else {

            arr[++top] = x;

            cout << x << " pushed into stack\n";

            return true;

        }

    }

    int pop() {

        if (top < 0) {

            cout << "Stack Underflow";

            return 0;

        }

        else {

            int x = arr[top--];

            return x;

        }

    }

    int peek() {

        if (isEmpty()) {

            cout << "Stack is Empty";

            return 0;

        }

        else {

            int x = arr[top];

            return x;

        }

    }

    bool isEmpty() { return (top < 0); }

    bool isFull() { return (top >= (MAX\_SIZE - 1)); }

};

int main()

{

    class Stack s;

    s.push(10);

    s.push(20);

    s.push(30);

    // print top element of stack after popping

    cout << "Top element is: " << s.peek() << endl;

    // print all elements in stack:

    cout << "Elements present in stack: ";

    while(!s.isEmpty()) {

        cout << s.peek() <<" ";

        s.pop();

    }

    return 0;

}

**Example 2**: Using linked list (🡪 dynamic stack)

#include <iostream>

class Node {

public:

    int data;

    Node\* next;

    Node(int value) {

        data = value;

        next = nullptr;

    }

};

class Stack {

private:

    Node\* top;

public:

    Stack() {

        top = nullptr;

    }

    void push(int element) {

        Node\* newNode = new Node(element);

        newNode->next = top;

        top = newNode;

    }

    int pop() {

        if (isEmpty()) {

            std::cout << "Stack underflow!" << std::endl;

            return -1; // Error value

        }

        Node\* temp = top;

        int topElement = top->data;

        top = top->next;

        delete temp;

        return topElement;

    }

    int peek() const {

        if (isEmpty()) {

            std::cout << "Stack is empty!" << std::endl;

            return -1; // Error value

        }

        return top->data;

    }

    bool isEmpty() const {

        return top == nullptr;

    }

    int size() const {

        int count = 0;

        Node\* current = top;

        while (current != nullptr) {

            count++;

            current = current->next;

        }

        return count;

    }

};

int main() {

    Stack stack;

    stack.push(10);

    stack.push(20);

    stack.push(30);

    std::cout << "Stack size: " << stack.size() << std::endl;

    while (!stack.isEmpty()) {

        std::cout << "Top element: " << stack.peek() << std::endl;

        stack.pop();

    }

    return 0;

}

**Example 3**: Using vector (🡪 dynamic stack)

#include <iostream>

#include <vector>

class Stack {

private:

    std::vector<int> data;

public:

    void push(const int& element) {

        data.push\_back(element);

    }

    int pop() {

        if (isEmpty()) {

            throw std::runtime\_error("Stack is empty");

        }

        int topElement = data.back();

        data.pop\_back();

        return topElement;

    }

    int top() const {

        if (isEmpty()) {

            throw std::runtime\_error("Stack is empty");

        }

        return data.back();

    }

    bool isEmpty() const {

        return data.empty();

    }

    size\_t size() const {

        return data.size();

    }

};

int main() {

    Stack stack;

    stack.push(10);

    stack.push(20);

    stack.push(30);

    std::cout << "Stack size: " << stack.size() << std::endl;

    while (!stack.isEmpty()) {

        std::cout << "Top element: " << stack.top() << std::endl;

        stack.pop();

    }

    return 0;

}

### Stack in C++ STL

<https://www.geeksforgeeks.org/stack-in-cpp-stl/>

## Queue

### Definition

Queue is a linear data structure that allows adding and removing elements in a specific order. To understand a queue, think of a cafeteria line: new people are added to the line at the back; the first person in line is served first, and the last person is served last. So, **in a queue the first item added to it will be the first item removed from it**. As a result, a queue is said to have "first in first out" behavior (or *FIFO*). That is opposite to a [stack](#_2et92p0).

*Note:*

Although the concept is simple, programming a queue is not as simple as programming a *stack*.

Let's go back to the example of the cafeteria line. Let's say one person leaves the line. Then what? Everyone in line must step forward, right? Now, imagine if only one person could move at a time. So, the second person steps forward to fill the space left by the first person, and then the third person steps forwards to fill the space left by the second person, and so on. Now imagine that no one can leave or be added to the line until everyone has stepped forward. You can see the line will move very slowly.

It is not difficult to program a queue that works, but it is **quite touch to make a queue that works fast**!

### Operations & Complexities

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Meaning | Time Complexity | Space Complexity |
| Q.enqueue(val) | Adds an item to a queue. If the queue is full, it is said to be an *Overflow* condition | O(1) | O(1) |
| Q.dequeue() | Removes an item from a queue. If the queue is empty, it is said to be an *Underflow* condition | O(1) | O(1) |
| Q.front() | Get the front item from a queue | O(1) | O(1) |
| Q.back() or Q.rear() | Get the last item from a queue | O(1) | O(1) |
| Q.empty() | Returns true if queue is empty, else false | O(1) | O(1) |
| Q.full() | Returns true if queue is full, else false | O(1) | O(1) |
| Q.size() | Get the numer of elements in a queue | O(1) | O(1) |



### Applications

Queue is used when things don’t have to be processed immediately, but have to be processed in FIFO order like [Breadth First Search](http://en.wikipedia.org/wiki/Breadth-first_search). This property makes queue useful in following scenarios.

* When a resource is shared among multiple consumers. For examples, CPU scheduling or disk scheduling.
* When data is transferred asynchronously (data isn’t necessarily received at same rate as sent) between two processes. For examples, IO buffers, pipes, file IO, sockets, etc.
* Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario.

### Implementation

There are some ways to implement a queue:

**Using array**:

* The first method is to make an array and shift all the elements to accommodate enqueues and dequeues. This is slow, because with many elements, the shifting takes time.
* The second method is, instead of shifting the elements, shifting the enqueue and dequeue points. Imagine that cafeteria line again. If the front of the line continually moves backward as each person leaves the line, then people don't need to step forward or backward, which saves time.???
* This method is much more complicated than the first one. Instead of keeping track of just the enqueue point (the "end"), we also need to keep track of the dequeue point (the "front"). This all gets even more complicated when we realize that after a bunch of enqueues and dequeues, the line will need to wrap around the end of the array. Think of the cafeteria line. As people enter and leave the line, the line moves farther and farther backwards, and eventually it will circle the entire cafeteria and end up at its original location.???

<https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/>

**Using linked list:**

Example:

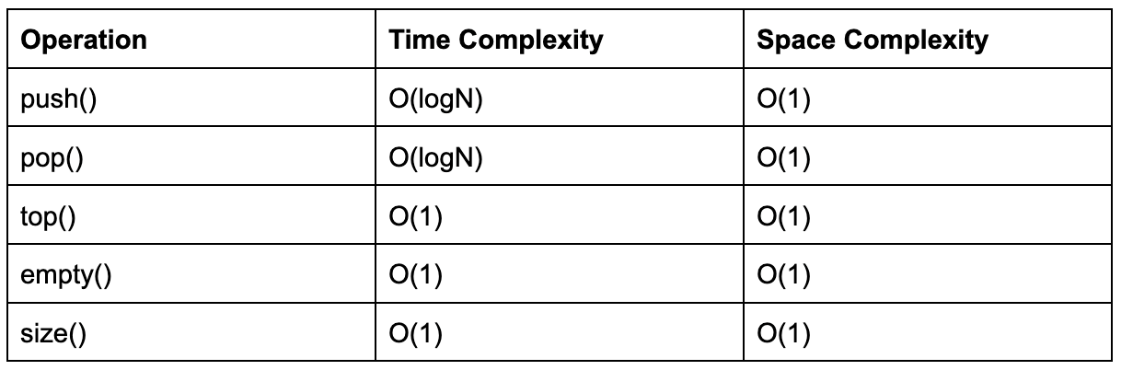
* <https://mathcs.clarku.edu/~fgreen/courses/cs170/FunctionPointers/Queue.c> (perfect, no memory leak)
* <https://gist.github.com/ArnonEilat/4471278> (can cause memory leak)

### Queue in C++ STL

<https://www.geeksforgeeks.org/queuepush-and-queuepop-in-cpp-stl/>

### Different Types of Queues

#### Priority Queue

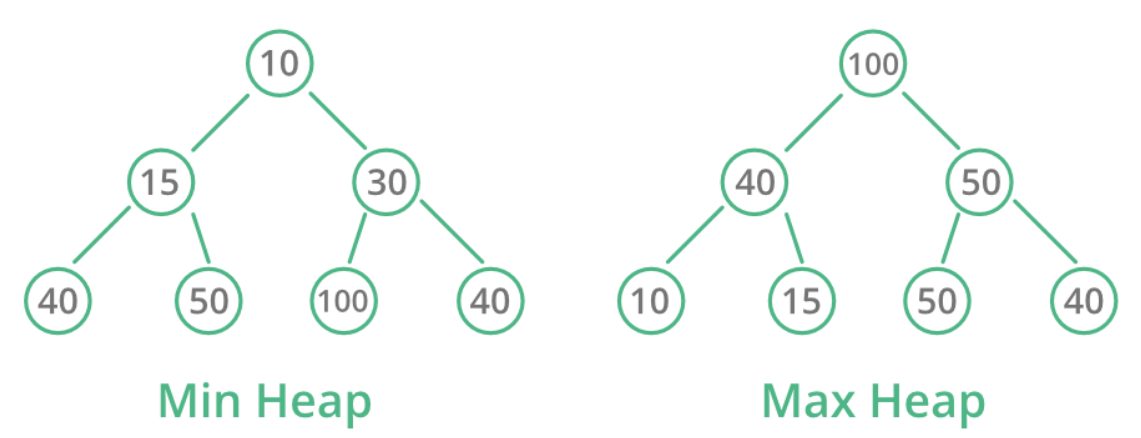


#### Circular Queue

## Heap

### Definition

A heap is a special **tree-based data structure** in which the tree is a complete [Binary Tree](#_Binary_Tree).



Types:

* **Max-Heap**: The **key present at the root node must be greatest** among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.
* **Min-Heap**: The **key present at the root node must be smallest** among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.

Other types:

1. [Binomial Heap](https://www.geeksforgeeks.org/binomial-heap-2/)
2. [Fibonacci Heap](https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/)
3. [Leftist Heap](https://www.geeksforgeeks.org/leftist-tree-leftist-heap/)
4. [K-ary Heap](https://www.geeksforgeeks.org/k-ary-heap/)

### Operations & Complexities

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Meaning | Time Complexity | Space Complexity |
| Heapify | Create a heap from an array. |  |  |
| Insertion | Insert an element in existing heap. | O(log N) |  |
| Deletion | Delete the top element of the heap or the highest priority element, and then organizing the heap and returning the element. | O(log N) |  |
| Peek | Find the top element of the heap. |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

### Applications

### Heap in C++ STL

## Pair

### Definition

The pair container is used to **combine together two values that may be of different data types**.

The first element is referenced as first and the second element as second and the order is fixed (first, second).

### Operations & Complexities

### Applications

### Pair in C++ STL

Example:

#include <iostream>

#include <utility> // for pair

int main()

{

    // Define a pair and assign value to it

    // Way 1:

    std::pair<int, char> mypair;  // defining a pair

    mypair.first = 1;         // first part of the pair

    mypair.second = 'a';     // second part of the pair

    // Way 2;

    // std::pair<int, char> mypair = {1, 'a'};

    // Way 3:

    // std::pair<int, char> mypair(1, 'a');

    // Way 4;

    // std::pair<int, char> mypair{1, 'a'};

    // Way 5;

    // std::pair<int, char> mypair = std::make\_pair(1, 'a');

    // Way 6:

    // std::pair<int, char> tmppair(1, 'a');

    // std::pair<int, char> mypair(tmppair);

    // Print the pair

    std::cout << mypair.first << " ";

    std::cout << mypair.second << std::endl;

    return 0;

}

Output:

1 a

## List

### Definition

List is a sequence container which **stores data in non-contiguous memory allocation**.

The idea of list is **similar to vectors** in C++. But compared to vectors, lists have low traversal, but once a position has been found, insertion and deletion are quick.

### Operations & Complexities

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Space Complexity |
| Find  Index | O(n) | O(1) |
| L.insert(it)  L.push\_back()  L.push\_front() | O(1) | O(1) |
| L.erase(it)  L.pop\_back()  L.pop\_front() | O(1) | O(1) |
| L.remove(val) | O(n) |  |
| L.size() | O(1) | O(1) |
| L.empty() | O(1) | O(1) |

### Applications

### List in C++ STL

Example:

#include <iostream>

#include <list> // for list

using namespace std;

void showlist(std::list<int> inList)

{

    std::list<int>::iterator it;

    for (it = inList.begin(); it != inList.end(); ++it) {

        cout << '\t' << \*it;

    }

}

int main()

{

    std::list<int> list1, list2;

    for (int i = 0; i < 10; ++i) {

        list1.push\_back(i);

        list2.push\_front(i);

    }

    cout << "\nlist1: ";

    showlist(list1);

    cout << "\nlist2: ";

    showlist(list2);

    cout << "\nlist1.front() : " << list1.front();

    cout << "\nlist1.back() : " << list1.back();

    cout << "\nlist1.pop\_front() : ";

    list1.pop\_front();

    showlist(list1);

    cout << "\nlist2.pop\_back() : ";

    list2.pop\_back();

    showlist(list2);

    cout << "\nlist1.reverse() : ";

    list1.reverse();

    showlist(list1);

    cout << "\nlist2.sort(): ";

    list2.sort();

    showlist(list2);

    return 0;

}

Output:

list1: 0 1 2 3 4 5 6 7 8 9

list2: 9 8 7 6 5 4 3 2 1 0

list1.front() : 0

list1.back() : 9

list1.pop\_front() : 1 2 3 4 5 6 7 8 9

list2.pop\_back() : 9 8 7 6 5 4 3 2 1

list1.reverse() : 9 8 7 6 5 4 3 2 1

list2.sort(): 1 2 3 4 5 6 7 8 9

### Notes

* List vs Vector: <https://www.geeksforgeeks.org/difference-between-vector-and-list/>

## Tuple

[Tuples in C++ - GeeksforGeeks](https://www.geeksforgeeks.org/tuples-in-c/)

[tie - C++ Reference (cplusplus.com)](https://cplusplus.com/reference/tuple/tie/)

## Set

### Definition

Set is a type of **associative container** in which each element value has to be unique because the value of the element identifies it.

Properties:

* All the elements in a set have **unique values**.
* Sets store elements in **sorted order** or **unsorted order**.

That’s why they are grouped into two categories: *ordered set* and *unorded set*.

* Values in a set are **immutable**. They cannot be modified once added to the set, though it’s possible to remove and then add the modified values of that element.
* Ordered sets follow the Binary search tree implementation, while unordered set follow Hash table implementation.

### Operations & Complexities

#### Ordered Set

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Space Complexity |
| S.find() | O(log n) | O(1) |
| S.insert(val) | O(log n) | O(1) |
| S.erase(val) | O(log n) | O(1) |
| S.erase(it) | O(1) | O(1) |
| S.size() | O(1) | O(1) |
| S.empty() | O(1) | O(1) |

#### Unordered Set

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Space Complexity |
| s.find() | O(1), or O(n) in worst case | O(1) |
| s.insert(val) | O(1), or O(n) in worst case | O(1) |
| s.erase(val) | O(1), or O(n) in worst case | O(1) |
| s.erase(it) | O(1), or O(n) in worst case | O(1) |
| s.size() | O(1) | O(1) |
| s.empty() | O(n) | O(1) |

### Applications

### Set in C++ STL

#### Ordered Set

#include <iostream>

#include <set>   // for ordered set

int main()

{

    // Way 1: Create and initialize an ordered set container by assignment

    // std::set<int> s1 = {40, 30, 60, 20, 50, 50, 10};

    // Way 2: Create and initialize an ordered set container by intializer list

    // std::set<int> s2{40, 30, 60, 20, 50, 50, 10};

    // Way 3: Create and initialize an ordered set container

    std::set<int> s1;

    s1.insert(40);

    s1.insert(30);

    s1.insert(60);

    s1.insert(20);

    s1.insert(50);

    s1.insert(50);  // Note: only one 50 will be added to the set

    s1.insert(10);

    // Print all elements of set s1

    std::set<int>::iterator it;

    std::cout << "\nThe set s1 is: \n";

    for (it = s1.begin(); it != s1.end(); it++) {

        std::cout << \*it << " ";

    }

    std::cout << std::endl;

    // Create another set called s2, then assign the elements from s1 to s2

    std::set<int> s2(s1.begin(), s1.end());

    // Print all elements of set ss

    std::cout << "\nThe set s2 is:\n";

    for (it = s2.begin(); it != s2.end(); it++) {

        std::cout << \*it << " ";

    }

    std::cout << std::endl;

    // Remove all elements less than 30 in set s2

    std::cout << "\nThe set s2 after removal of elements less than 30:\n";

    s2.erase(s2.begin(), s2.find(30));

    for (it = s2.begin(); it != s2.end(); it++) {

        std::cout << \*it << " ";

    }

    std::cout << std::endl;

    // Remove element with value 50 in set s2

    int num = s2.erase(50);

    std::cout << "\nThe set s2 after removal of the element 50:\n";

    for (it = s2.begin(); it != s2.end(); it++) {

        std::cout << \*it << " ";

    }

    std::cout << "\n => " << num << " element is removed\n";

    std::cout << std::endl;

    return 0;

}

Output:

The set s1 is:

10 20 30 40 50 60

The set s2 is:

10 20 30 40 50 60

The set s2 after removal of elements less than 30:

30 40 50 60

The set s2 after removal of the element 50:

30 40 60

 => 1 element is removed

#### Unordered Set

#include <iostream>

#include <unordered\_set> // for unordered set

int main()

{

    // Define an unordered set for storing string data-type

    std::unordered\_set<std::string> s ;

    // Insert various strings. Same string will be stored once in set

    s.insert("code") ;

    s.insert("in") ;

    s.insert("c++") ;

    s.insert("is") ;

    s.insert("fast") ;

    // Find and return if the key is found

    std::string key = "slow" ;

    if (s.find(key) == s.end()) {

        std::cout << "The key '" << key << "' " << "is not found" << std::endl;

    }

    else {

        std::cout << "Found " << key << std::endl;

    }

    // Print all elements of the set

    std::cout << "All elements: ";

    std::unordered\_set<std::string>::iterator it;

    for (it = s.begin(); it != s.end(); it++) {

        std::cout << \*it << " ";

    }

}

Output:

The key 'slow' is not found

All elements: fast is c++ in code

#### Notes

**1.** If ordered set stores user-defined data type (class or struct), have to overload operator **<** so the set knows how to sort its items.

Other ways: <https://stackoverflow.com/a/46128321/14835442>

**2.** There’re some ways to find item from set:

1. set::find(): It uses overload operator <. So if set stores user-defined data type, have to overload operator <.
2. std::find(): It uses overload operator ==. So if set stores user-defined data type, have to overload operator ==.
3. std::find\_if(): It doesn’t uses overload operator. So we don’t have to overload any operator, but have to write a comparision function to pass to find\_if().
4. Manually iterate each item and write your own code.

**3.** Operator overloads work with **objects, not pointers**. So if your set is "std::set<object\*>", it cannot work with operator overload.

The solution is to create custom comparator function or functor, and pass it to the set. This is similar to "[Other ways](https://stackoverflow.com/a/46128321/14835442)" in note 1 (just replace object with pointer).

**4**. Don't increase the iterator when ERASING element:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<int> s{ 1, 2, 3, 4, 5 };

    set<int>::iterator it;

    // BAD WAY

    //  In Visual Studio, this code will crash

// with exception "Expression: cannot increment value-initialized map/set iterator"

    //   because we tried to increase "it" even when it is "end"

    int i = 0;

    for (it = s.begin(); it != s.end(); ++it) {

    if (i == 4) {

        it = s.erase(it); // After erase, "it" will be increase by 1

// So when "it" is "end", "++it" in the loop will cause exception

    }

    i++;

    }

    // GOOD WAY

    //  Advice here: If you erase, you should not increase the iterator (more [details](https://stackoverflow.com/questions/72869355/stdset-using-iterator-causes-memory-violation-exception))

    int i = 0;

    for (it = s.begin(); it != s.end(); ) {

        if (i == 4)  {

            it = s.erase(it);    // After erase, "it" will be increase by 1

        }

        else {

            it++;                // Increase "it" by 1

        }

        i++;

    }

    // Print all elements in the set

    for (auto x : ms) {

        cout << x << " ";

    }

    return 0;

}

Output:

1 2 3 4

## Multi-Set

### Definition

Multiset is a type of associative containers. It’s similar to the set in all properties, except that multiple elements can have the **same values**.

### Operations & Complexities

#### Ordered Multiset

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Space Complexity |
| s.find() | O(log n) | O(1) |
| s.insert(val) | O(log n) | O(1) |
| s.erase(val) | O(log n) | O(1) |
| s.erase(it) | O(1) | O(1) |
| s.size() | O(1) | O(1) |
| s.empty() | O(1) | O(1) |

#### Unordered Multiset

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Space Complexity |
| s.find() | O(1), or O(n) in worst case | O(1) |
| s.insert(val) | O(1), or O(n) in worst case | O(1) |
| s.erase(val) | O(1), or O(n) in worst case | O(1) |
| s.erase(it) | O(1), or O(n) in worst case | O(1) |
| s.size() | O(1) | O(1) |
| s.empty() | O(n) | O(1) |

### Applications

### Multi-Set in C++ STL

#### Ordered Multiset

#include <iostream>

#include <set> // for ordered multiset

int main()

{

    // Way 1: Create and initialize an ordered multiset container by assignment

    // std::multiset<int> ms1 = {40, 30, 60, 20, 50, 50, 10};

    // Way 2: Create and initialize an ordered multiset container by intializer list

    // std::multiset<int> ms1{40, 30, 60, 20, 50, 50, 10};

    // Way 3: Create an ordered multiset container

    std::multiset<int> ms1;

    ms1.insert(40);

    ms1.insert(30);

    ms1.insert(60);

    ms1.insert(20);

    ms1.insert(50);

    ms1.insert(50);  // Note: 50 will be added again

    ms1.insert(10);

    // Print all elements in the multiset ms1

    std::multiset<int>::iterator it;

    std::cout << "\nThe multiset ms1 is:\n";

    for (it = ms1.begin(); it != ms1.end(); ++it) {

        std::cout << \*it << " ";

    }

    std::cout << std::endl;

    // Create another multiset called ms2, then assign all elements from ms1 to ms2

    std::multiset<int> ms2(ms1.begin(), ms1.end());

    // Print all elements in the multiset ms2

    std::cout << "\nThe multiset ms2 is:\n";

    for (it = ms2.begin(); it != ms2.end(); ++it) {

        std::cout << \*it << " ";

    }

    std::cout << std::endl;

    // Remove all elements less than 30 in ms2

    std::cout << "\nThe multiset ms2 after removal of elements less than 30:\n";

    ms2.erase(ms2.begin(), ms2.find(30));

    for (it = ms2.begin(); it != ms2.end(); ++it) {

        std::cout << \*it << " ";

    }

    std::cout << std::endl;

    // Remove all elements with value 50 in ms2

    int num = ms2.erase(50);

    std::cout << "\nThe multiset ms2 after removal of the element 50:\n";

    for (it = ms2.begin(); it != ms2.end(); ++it) {

        std::cout << \*it << " ";

    }

    std::cout << std::endl;

    return 0;

}

Output:

The multiset ms1 is:

10 20 30 40 50 50 60

The multiset ms2 is:

10 20 30 40 50 50 60

The multiset ms2 after removal of elements less than 30:

30 40 50 50 60

The multiset ms2 after removal of the element 50:

30 40 60

#### Unordered Multiset

#include <iostream>

#include <unordered\_set>   // for unordered multiset

int main()

{

    // Create and initialize an unordered multiset by assignment

    std::unordered\_multiset<int> ums1 = {2, 7, 2, 5, 0, 3, 7, 5};

    // Create and initialize an unordered multiset by intializer list

    std::unordered\_multiset<int> ums2 ({1, 3, 1, 7, 2, 3, 4, 1, 6});

    // Print all elements in ums1

    std::cout << "\nThe multiset ums1 is:\n";

    std::unordered\_multiset<int>::iterator it;

    for (it = ums1.begin(); it != ums1.end(); ++it) {

        std::cout << \*it << " ";

    }

    std::cout << std::endl;

    // Print all elements in ums2

    std::cout << "\nThe multiset ums2 is:\n";

    for (it = ums2.begin(); it != ums2.end(); ++it) {

        std::cout << \*it << " ";

    }

    std::cout << std::endl;

    // Insert an element with value 7 to ums1

    ums1.insert(7);

    // Print all elements in ums1

    std::cout << "\nThe multiset ums1 after insertion of element with value 7 is:\n";

    for (it = ums1.begin(); it != ums1.end(); ++it) {

        std::cout << \*it << " ";

    }

    std::cout << std::endl;

    // Find and return iterator to first position of val if exist.

// Otherwise, return iterator to end

    int val = 3;

    if (ums1.find(val) != ums1.end()) {

        std::cout << "\nThe multiset ums1 contains " << val << std::endl;

    }

    else {

        std::cout << "\nThe multiset ums1 does not contains " << val << std::endl;

    }

    // Count and return total number of occurrence in set

    val = 5;

    int cnt = ums1.count(val);

    std::cout << std::endl << val << " appears " << cnt << " times in multiset ums1 \n";

    // Deletes all elements with value of val

    ums2.erase(val);

    // Print all elements in ums2

    std::cout << "\nThe multiset ums2 after removal of element with value " << val << " is:\n";

    for (it = ums2.begin(); it != ums2.end(); ++it) {

        std::cout << \*it << " ";

    }

    std::cout << std::endl;

}

Output:

The multiset ums1 is:

3 0 5 5 7 7 2 2

The multiset ums2 is:

6 4 2 7 3 3 1 1 1

The multiset ums1 after insertion of element with value 7 is:

3 0 5 5 7 7 7 2 2

The multiset ums1 contains 3

5 appears 2 times in multiset ums1

The multiset ums2 after removal of element with value 5 is:

6 4 2 7 3 3 1 1 1

* **Selecting between ordered and unordered associative containers:**

<https://embeddedartistry.com/blog/2017/08/30/choosing-the-right-container-associative-containers/>

## Map

### Definition

Map is a type of associative container which stores elements in a combination of key-value.

Properties:

* Each element has a key and a value.
* All the elements in a map have **unique key**. In other words, no two values can have the same key.
* Maps store elements in **sorted order** or **unsorted order**.

That’s why maps are grouped into two categories: *ordered map* and *unorded map*.

* Ordered maps follow the Binary search tree implementation, while unordered maps follow Hash table implementation.
* A map is actually a **list of pairs**.

### Operations & Complexities

#### Ordered Map

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Space Complexity |
| M.find(val) | O(log n) | O(1) |
| M.insert(pair<int, int> (val1, val2) | O(log n) | O(1) |
| M.erase(val) | O(log n) | O(1) |
| M.erase(it) | O(1) | O(1) |
| M.empty( ) | O(1) | O(1) |
| M.clear( ) | O(n) | O(1) |
| M.size( ) | O(1) | O(1) |

#### Unordered Map

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Space Complexity |
| M.find(val) | O(1), or O(n) in worst case | O(1) |
| M.insert(pair<int, int> (val1, val2) | O(1), or O(n) in worst case | O(1) |
| M.erase(val) | O(1), or O(n) in worst case | O(1) |
| M.erase(it) | O(1), or O(n) in worst case | O(1) |
| M.empty( ) | O(1) | O(1) |
| M.clear( ) | O(n) | O(1) |
| M.size( ) | O(1) | O(1) |

### Applications

### Map in C++ STL

#### Ordered Map

#include <iostream>

#include <map>      // for ordered map

int main()

{

    // Create an ordered map

    std::map<int, int> m1;

    m1[1] = 40;

    m1[2] = 30;

    m1[3] = 60;

    m1[4] = 20;

// Another way of inserting a value in a map

    m1.insert(std::pair<int, int>(7, 10));

    m1.insert(std::pair<int, int>(5, 50));   // Note: Element of key 5 and element of key 7 will be reverted because m is an ordered map in accessending

    m1.insert(std::pair<int, int>(6, 50));

m1.insert(std::pair<int, int>(6, 90)); // Note: Inserting element with the same key as previous elements is ignored

    // Print all elements in map m1

    std::map<int, int>::iterator it;

    std::cout << "\nThe map m1 is:\n";

    std::cout << "\tKEY\tVALUE\n";

    for (it = m1.begin(); it != m1.end(); ++it) {

        std::cout << '\t' << it->first << '\t' << it->second << std::endl;

    }

    // Create another map called m2, then assign all elements from m1 to m2

    std::map<int, int> m2(m1.begin(), m1.end());

    // Print all elements in map m1

    std::cout << "\nThe map m2 is:\n";

    std::cout << "\tKEY\tVALUE\n";

    for (it = m2.begin(); it != m2.end(); ++it) {

        std::cout << '\t' << it->first << '\t' << it->second << std::endl;

    }

    // Remove all elements with keys smaller than 3 in m2

    std::cout << "\nThe map m2 after removal of elements with keys smaller than 3:\n";

    std::cout << "\tKEY\tVALUE\n";

    m2.erase(m2.begin(), m2.find(3));

    for (it = m2.begin(); it != m2.end(); ++it) {

        std::cout << '\t' << it->first << '\t' << it->second << std::endl;

    }

    int num = m2.erase(4);

    std::cout << "\nThe map m2 after removal of element at key 4:\n";

    std::cout << "  ==> " << num << " element is removed\n";

    std::cout << "\tKEY\tVALUE\n";

    for (it = m2.begin(); it != m2.end(); ++it) {

        std::cout << '\t' << it->first << '\t' << it->second << std::endl;

    }

    return 0;

}

Output:

The map m1 is:

KEY VALUE

1 40

2 30

3 60

4 20

5 50

6 50

7 10

The map m2 is:

KEY VALUE

1 40

2 30

3 60

4 20

5 50

6 50

7 10

The map m2 after removal of elements with keys smaller than 3:

KEY VALUE

3 60

4 20

5 50

6 50

7 10

The map m2 after removal of element at key 4:

==> 1 element is removed

KEY VALUE

3 60

5 50

6 50

7 1

#### Unordered Map

#### Notes

**1.** If ordered map has key of user-defined data type (class or struct), have to overload operator **<** so the map knows how to sort its items.

Other ways: <https://stackoverflow.com/a/46128321/14835442> (samples are about set, but can apply similarly to map)

**2.** There’re some ways to find item from map:

1. map::find(): It uses overload operator <. So if set stores user-defined data type, have to overload operator <.
2. std::find\_if(): It doesn’t uses overload operator. So we don’t have to overload any operator, but have to write a comparision function to pass to find\_if().
3. Write your own searching function.

Note: std::find() doesn’t work with map.

**3.** Operator overloads work with **objects, not pointers**. So if your map is "std::map<object1\*, object2>", then it cannot work with operator overload.

The solution is to create custom comparator function or functor, and pass it to the map. This is similar to "[Other ways](https://stackoverflow.com/a/46128321/14835442)" in note 1 (just replace object with pointer).

**4.** The std::sort() won’t work with map. So, to sort items in map, we can do one of following workaround ways:

1. Make another map, which uses the original map’s values as its keys and the original map’s keys as its values. A multimap is used because values of the original map can be duplicate.
2. Copy the map into a vector (or set, list, etc.) of key-value pairs. Then sort items in the vector using std::sort().
3. Write your own sorting function.

Examples [here](https://www.educative.io/answers/how-to-sort-a-map-by-value-in-cpp).

Note: STL map doesn’t provide map::sort().

## Multi-Map

### Definition

Multimap is a type of associative containers. It’s similar to the map in all properties, except that multiple elements can have the **same keys**.

### Operations & Complexities

#### Ordered Map

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Space Complexity |
| M.find(val) | O(log n) | O(1) |
| M.insert(pair<int, int> (val1, val2) | O(log n) | O(1) |
| M.erase(val) | O(log n) | O(1) |
| M.erase(it) | O(1) | O(1) |
| M.empty( ) | O(1) | O(1) |
| M.clear( ) | O(n) | O(1) |
| M.size( ) | O(1) | O(1) |

#### Unordered Map

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Space Complexity |
| M.find(val) | O(1), or O(n) in worst case | O(1) |
| M.insert(pair<int, int> (val1, val2) | O(1), or O(n) in worst case | O(1) |
| M.erase(val) | O(1), or O(n) in worst case | O(1) |
| M.erase(it) | O(1), or O(n) in worst case | O(1) |
| M.empty( ) | O(1) | O(1) |
| M.clear( ) | O(n) | O(1) |
| M.size( ) | O(1) | O(1) |

### Applications

### Multi-Map in C++ STL

#### Ordered Map

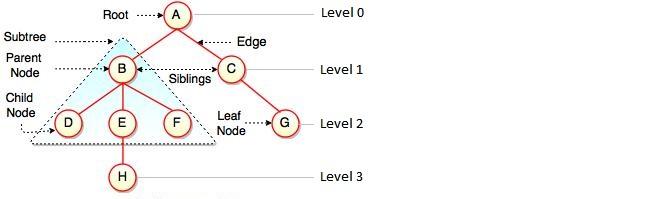
#### Unordered Map

## Tree

### General Tree

#### Definition

Tree is a non-linear data structure which represents the nodes connected by edges. It’s used to store the information in the form of **hierarchy style**.



Terms:

* *Path* – Represents the sequence of nodes along the edges of a tree.
* *Visiting* – Represents checking the value of a node when control is on the node.
* *Traversing* – Represents passing through nodes in a specific order.
* *Keys* − Represents a value of a node based on which a search operation is to be carried out for a node.
* *Height of Tree* – Represents the height of its root node.
* *Height of Node* – Represents the number of edges on the longest path between that node and a leaf.
* *Depth of Node* – Represents the number of edges from the tree's root node to the node.
* *Degree of Node* – Represents a number of children of a node.

#### Traversal

In order to process trees, we need a mechanism for traversing them. Each node is processed only once but it may be visited more than once. As we have already seen **in linear data structures (like linked lists, stacks, queues, etc.), the elements are visited in sequential order. But, in tree structures there are many different ways**.

Starting at the root of a BT, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps are: performing an action on the current node (denoted with "D"), traversing to the left child node (denoted with "L"), and traversing to the right child node (denoted with "R"). This process can be easily described through recursion.

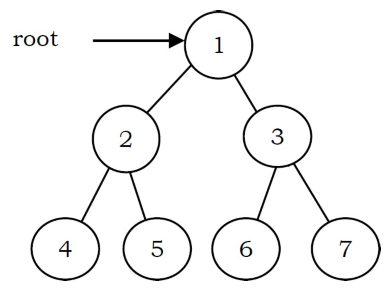
*So, we can classify the traversals into several styles:*

* *In-order traversal (LDR)*
* *Pre-order traversal (DLR)*
* *Post-order traversal (LRD)*

*There is another traversal method which does not depend on the above orders. It is:*

* *Level-order traversal*

Let us use the diagram below for the remaining discussion.



##### In-Order Traversal (LDR)

The nodes of tree would be visited in the order: 4 2 5 1 6 3 7

In in-order traversal, the root is visited between the subtrees. It is defined as follows:

1. Traverse the left subtree in in-order.
2. Visit the root.
3. Traverse the right subtree in in-order.

*Time Complexity: O(n).*

*Space Complexity: O(n).*

##### Pre-Order Traversal (DLR)

The nodes of tree would be visited in the order: 1 2 4 5 3 6 7

In pre-order traversal, each node is processed before (pre-) either of its subtrees. It is defined as follows:

1. Visit the root.
2. Traverse the left subtree in pre-order.
3. Traverse the right subtree in pre-order.

*Time Complexity: O(n).*

*Space Complexity: O(n).*

##### Post-Order Traversal (LRD)

The nodes of the tree would be visited in the order: 4 5 2 6 7 3 1

In post-order traversal, the root is visited after both subtrees. It is defined as follows:

1. Traverse the left subtree in post-order.
2. Traverse the right subtree in post-order.
3. Visit the root.

*Time Complexity: O(n).*

*Space Complexity: O(n).*

##### Level-Order Traversal

The nodes of the tree are visited in the order: 1 2 3 4 5 6 7

Level-order traversal is defined as follows:

1. Visit the root (level 0).
2. Visit all nodes at level 1, from left to right.
3. Go to the next level and visit all the nodes at that level.
4. Repeat this until all levels are completed.

*Time Complexity: O(n).*

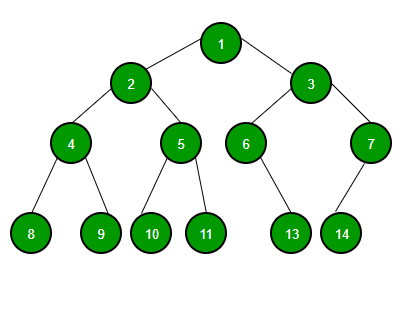
*Space Complexity: O(n). Since, in the worst case, all the nodes on the entire last level could be in the queue simultaneously.*

### Binary Tree

#### Definition

BT is a special tree used for data storage purposes. It has a special condition that **each node can have a maximum of two children**.

A BT has the benefits of both an ordered array and a linked list – **Search in BT is as quick as in a sorted array, and insertion or deletion in BT are as fast as in linked list**.



#### Operations & Complexities

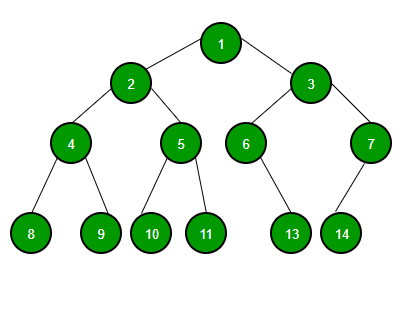
|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** |  | O(n) | O(n) |
| **Insert** |  | O(n) | O(n) |
| **Delete** |  | O(n) | O(n) |

Where: n is number of nodes

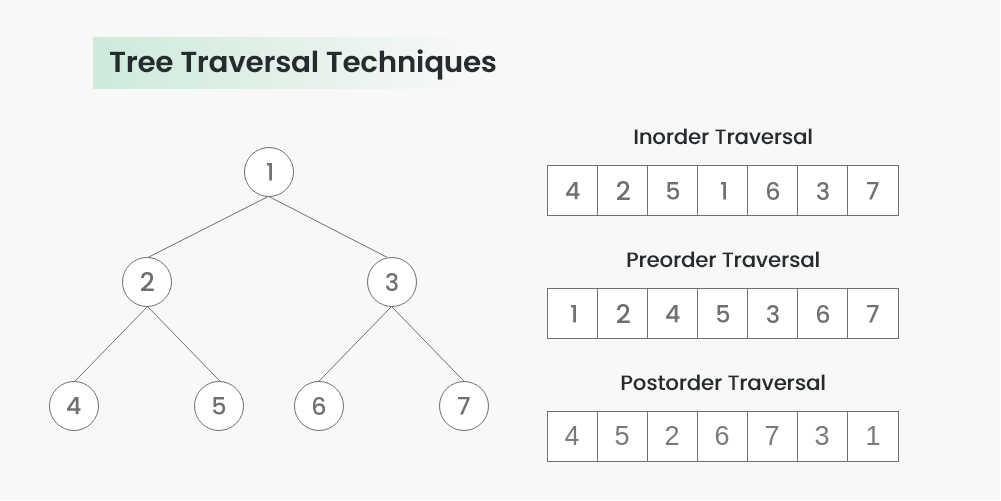
#### Definition

BT is a special tree used for data storage purposes. It has a special condition that **each node can have a maximum of two children**.

A BT has the benefits of both an ordered array and a linked list – **Search in BT is as quick as in a sorted array, and insertion or deletion in BT are as fast as in linked list**.



#### Traversal



##### In-Order Traversal (LDR)

Used algorithm: Depth First Search (DFS)

|  |  |
| --- | --- |
|  | 4  2  5  1  6  3  7 |

##### Pre-Order Traversal (DLR)

Used algorithm: Depth First Search (DFS)

1 2 4 5 3 6 7

##### Post-Order Traversal (LRD)

Used algorithm: Depth First Search (DFS)

4 5 2 6 7 3 1

##### Level-Order Traversal

Used algorithm: Breadth First Search (BFS)

<https://www.geeksforgeeks.org/level-order-tree-traversal/>

#### Implementation

##### In-Order Traversal (LDR)

#include <iostream>

using namespace std;

struct Node {

int data;

struct Node\* left, \* right;

};

Node\* newNode(int data) {

Node\* temp = new Node;

temp->data = data;

temp->left = NULL;

temp->right = NULL;

return temp;

}

void printInorder(struct Node\* node) {

if (node == NULL) {

return;

}

// First recur on left child

printInorder(node->left);

// Then print the data of node

cout << node->data << " ";

// Now recur on right child

printInorder(node->right);

}

// Driver code

int main() {

struct Node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

root->right->left = newNode(6);

root->right->right = newNode(7);

// Function call

cout << "Inorder traversal of binary tree is \n";

printInorder(root);

return 0;

}

Output:

4 2 5 1 6 3 7

##### Pre-Order Traversal (DLR)

...

void printInorder(struct Node\* node) {

if (node == NULL) {

return;

}

cout << node->data << " ";

printInorder(node->left);

printInorder(node->right);

}

...

Output:

1 2 4 5 3 6 7

##### Post-Order Traversal (LRD)

...

void printInorder(struct Node\* node) {

if (node == NULL) {

return;

}

printInorder(node->left);

printInorder(node->right);

cout << node->data << " ";

}

...

Output:

4 5 2 6 7 3 1

##### Level-Order Traversal

#### Applications

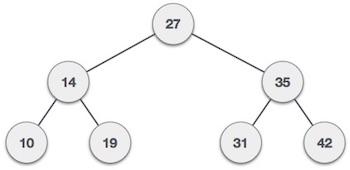
Following is the some of the applications where BT plays an important role:

* Expression trees used in compilers.
* Huffman coding trees used in data compression algorithms.
* [Binary Search Tree](#_1fob9te) (BST), which supports search, insertion and deletion on a collection of items in O(logn) (average).
* [Priority Queue](#_gjdgxs) (PQ), which supports search and deletion of minimum (or maximum) on a collection of items in logarithmic time (worst case).

### Binary Search Tree

#### Definition

BST exhibits a special behavior of a binary tree. **A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value**. Also, equal node values are not allowed in BST.



#### Operations & Complexities

Complexity for all BST operations depends on BST height (h). It's like:

|  |  |
| --- | --- |
| Worst case | Best case |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** | - Start at root  - Take left and right children as necessary | O(logh) | O(h) |
| **Insert** | - Similar to search, but look for null pointer | O(logh) | O(h) |
| **Delete** | - Search (if necessary)  - Different cases based on tree degree (number of children):  + 0 (meaning it’s a leaf node): just delete  + 1: similar to doubly linked list  + 2: swap with max (left) and delete (0 or 1 children)  - Update size and root | O(logh) | O(h) |

##### Tree Rotations

* Based around a pivot node (\*)
* Two directions: left and right

|  |  |
| --- | --- |
| * Left rotation: * Parent becomes left child * Pivot becomes parent * Old left child becomes parent right child * Pivot is new "root" | * Right rotation:   + Parent becomes right child   + Pivot becomes parent   + Old right child becomes parent left child   + Pivot is new "root" |

##### Insertion

##### Deletion

#### Traversal

#### Implementation

#### Applications

### Balanced Binary Search Tree

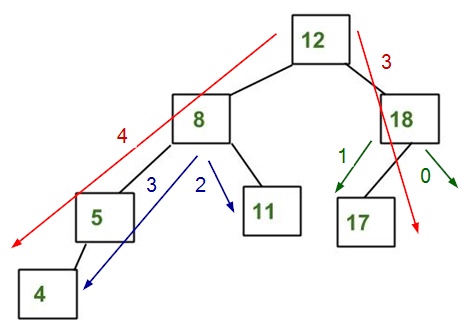
Balanced BST is a BST with self-balancing capability. It is divided into following subtypes:

#### AVL Tree

##### Definition

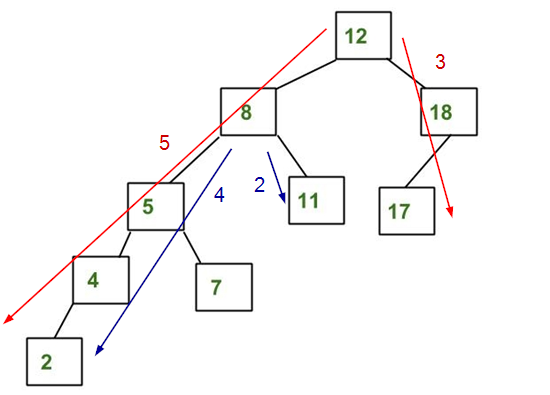
AVL tree (named after inventors Adelson-Velsky and Landis) is a self-balancing BST where the **difference between heights of left and right subtrees cannot be more than one for all nodes**. The balance is maintained using *tree rotations* as nodes inserted and deleted.

Example: AVL Tree



It is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

Example: NOT AVL Tree



It is not AVL because differences between heights of left and right subtrees for 8 is 4 (greater than 1) and for 12 is 5 (greater than 1).

##### Operations & Complexities

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** |  | O(logh) | O(logh) |
| **Insert** |  | O(logh) | O(logh) |
| **Delete** |  | O(logh) | O(logh) |

Where: h is height of tree

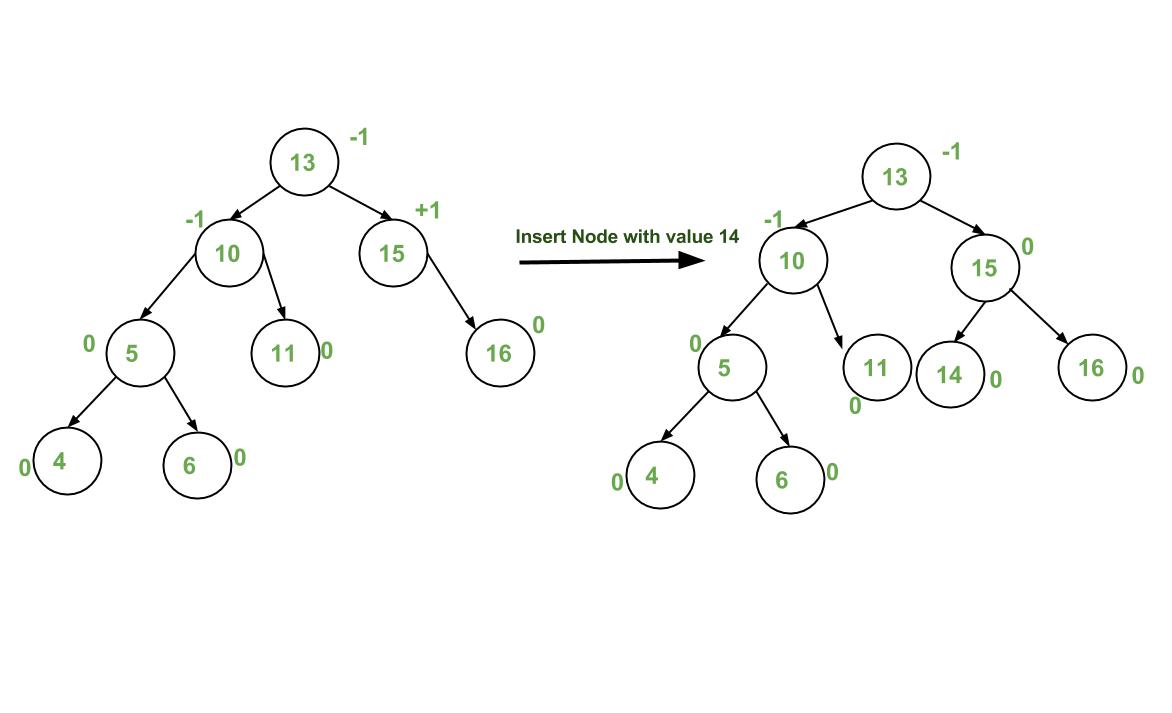
*More details:*

Most of the BST operations take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed binary tree. If we **make sure that height of the tree remains O(logn)** after every insertion and deletion, then we can guarantee an upper bound of O(logn) for all these operations. The height of an AVL tree is always O(logn) where n is the number of nodes in the tree.

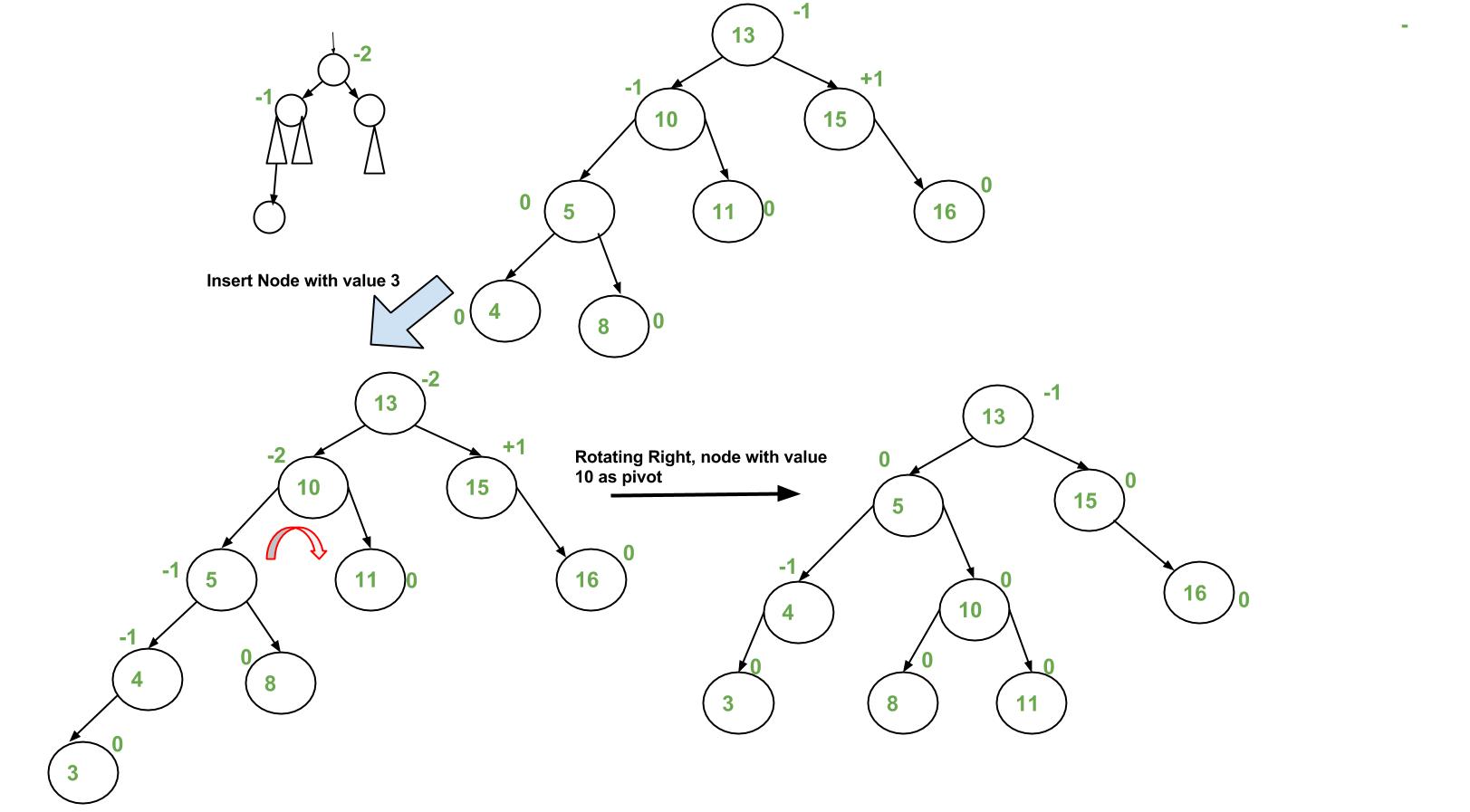
###### Insertion

Every time a new node is inserted, the tree must be re-balanced (if needed). One of following five cases can happen:

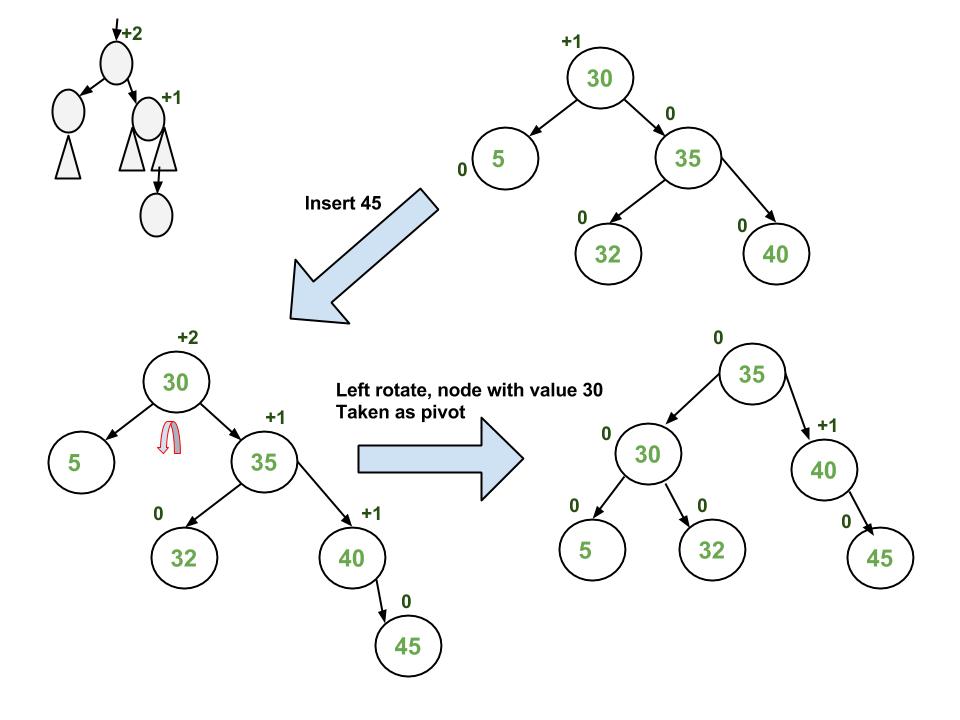
Case 1: No rotation needed



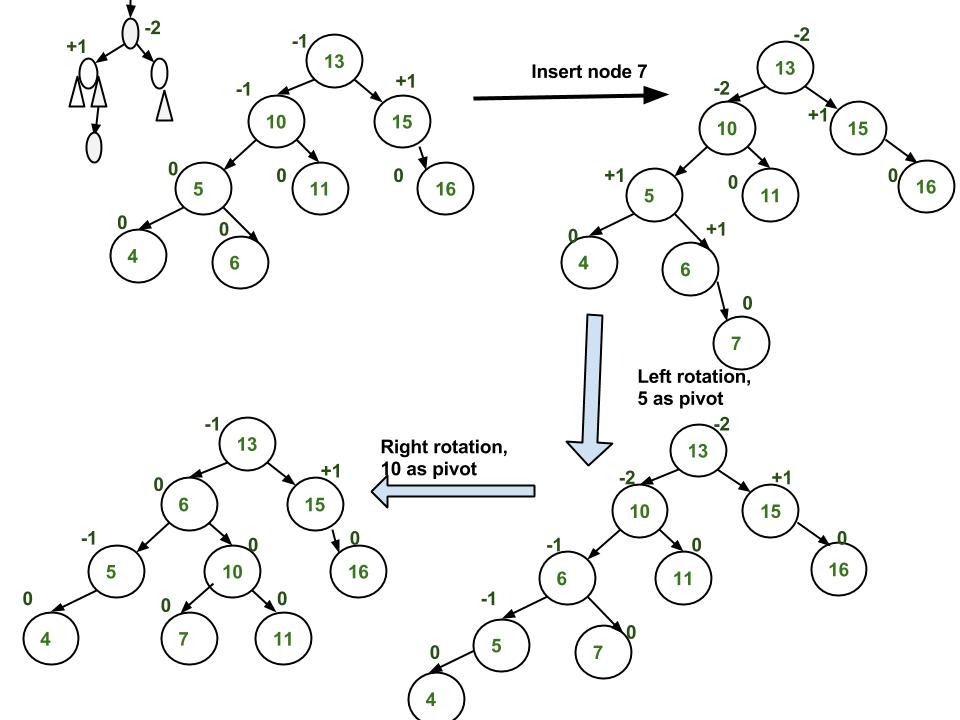
 Case 2: Right rotation



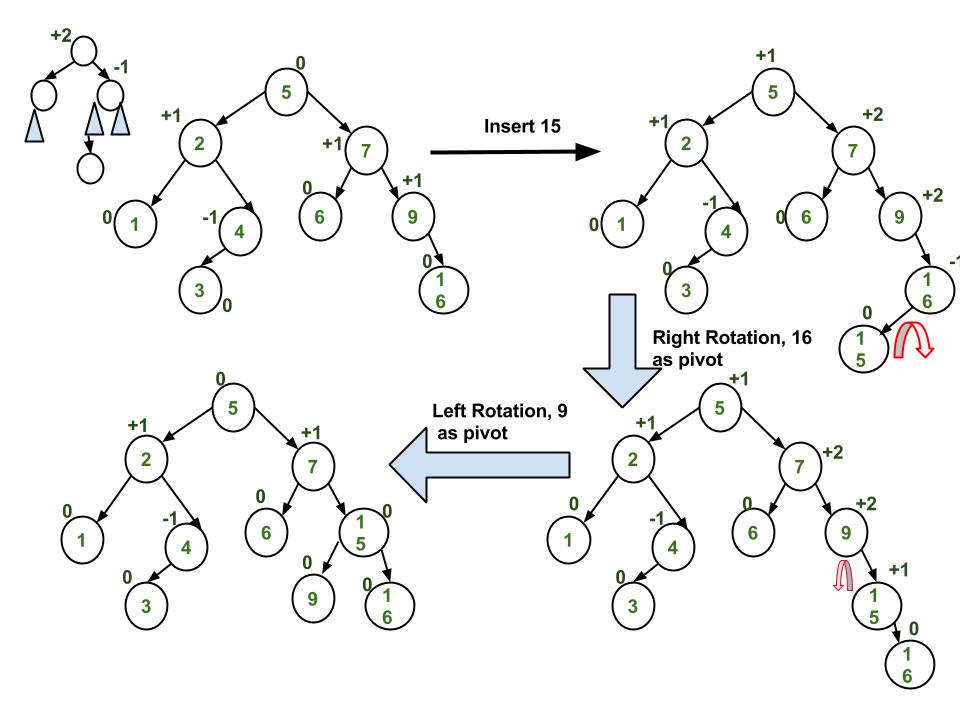
Case 3: Left rotation



Case 4: Left rotation, then right rotation



Case 5: Right rotation, then left rotation



Implementation: <https://www.geeksforgeeks.org/avl-tree-set-1-insertion/>

###### Deletion

Every time a node is deleted, the tree must be re-balanced (if needed). And there are five cases can happen, just similar to insertion.

Implementation: <https://www.geeksforgeeks.org/avl-tree-set-2-deletion/>

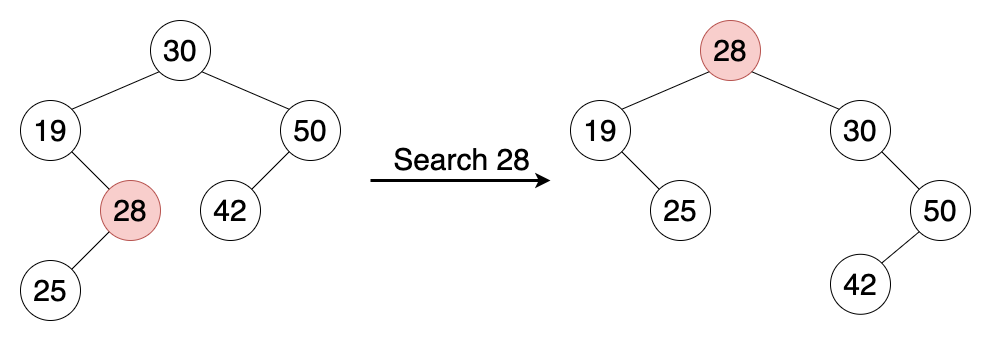
##### Applications

* Used in situations where frequent insertions are involved.
* Used in Memory management subsystem of the Linux kernel to search memory regions of processes during preemption.

#### Splay Tree

##### Definition

A splay tree is a self-balancing BST. After performing a search, insertion or deletion, splay trees perform an action called *splaying* where the tree is rearranged (using rotations) so that the **particular element is placed at the root of the tree**.



##### Basic Operations & Time Complexities

##### Applications

* Used to implement caches
* Used in garbage collectors.
* Used in data compression

#### Red-Black Tree

##### Definition

##### Operations & Complexities

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** |  | O(logh) | O(logh) |
| **Insert** |  | O(logh) | O(logh) |
| **Delete** |  | O(logh) | O(logh) |

Where: h is height of tree

##### Applications

* As a base for data structures used in computational geometry.
* Used in the Completely Fair Scheduler used in current Linux kernels.
* Used in the epoll system call implementation of Linux kernel.

#### Comparions

**AVL Tree vs. Red Black Tree**

The AVL tree and other self-balancing search trees like Red Black are useful to get all basic operations done in O(log n) time. The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So, if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is the more frequent operation, then AVL tree should be preferred over Red Black Tree.

**AVL Tree vs. Splay Tree**

Splay trees are simpler compared to AVL and Red-Black Trees as no extra field is required in every tree node. However, unlike AVL tree, a splay tree can change even with read-only operations like search.

### Balanced Search Tree

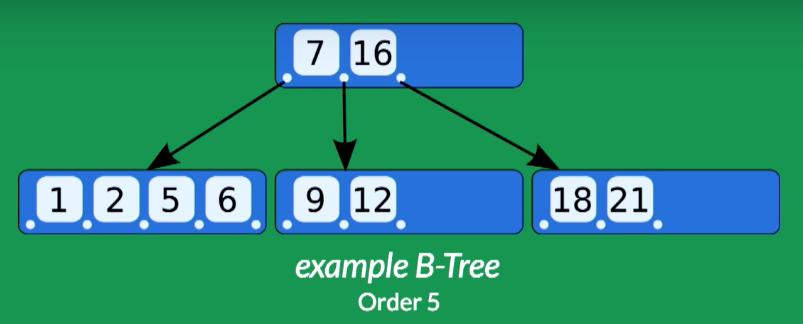
#### B-Tree

##### Definition

Should watch it first: <https://www.youtube.com/watch?v=C_q5ccN84C8&ab_channel=FullstackAcademy>

B-Tree is a self-balancing search tree, which contains **multiple nodes storing data in sorted order**. Each node can have **multiple children** and consists of **multiple keys**.

Example:



**Root node**

**Child node**

**Key**

The above B-Tree has:

* 1 root containing 2 keys: 7 and 16
* 3 child nodes. The left most node containing keys whose values are < 7. The middle node containing keys whose values are > 7 and < 16. The right most node containing keys whose values are > 16.
* Order of 5 (because its node – the left most one – has at most 5 children)

Properties:

A B-Tree of order **m** has:

* All leaves appear in the same level.
* Every node has at most **m** children.
* A non-leaf node with **k** children contains **k-1** keys.
* The root has at least two children if it is not a left node.
* Every non-left node (except the root) has at least a **ceiling of m/2** children.

##### Basic Operations & Time Complexities

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** |  | O(logh) | O(logh) |
| **Insert** |  | O(logh) | O(logh) |
| **Delete** |  | O(logh) | O(logh) |

Where: h is height of tree

**Insertion: 7:30**

##### Applications

* Used in database indexing to speed up the search.
* Used in file systems to implement directories.

*You might not know!*

To understand the use of B-Trees, we must think of the huge amount of data which cannot fit in main memory. So, we have to put the data in disk. But accessing and reading data from disk takes much more time than from main memory.

In this case, we can take advantage of B-Trees to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, etc.) require O(h) disk accesses where h is the height of the tree. But the height of B-Trees is kept low by putting maximum possible keys in a B-Tree node, so total disk accesses for most of the operations are reduced significantly compared to balanced BST (like AVL Tree, Red-Black Tree, etc.).

To conclude, **whenever you deal with some kind of external memory and the time to access the data of a node greatly exceeds the time spent processing that data (such as big databases)**, consider using B-Trees.

## Graph

### Definition

A graph is a non-linear data structure **consisting of nodes (or vertices) and edges** which are lines/arcs connecting any pair of nodes.

Mathematically, a graph G = (V, E) is a set of vertices V and edges E where each edge (u, v) is a connection between vertices: u, v ∈ V.

For example:



The above graph has a set of vertices V = {0, 1, 2, 3, 4} and a set of edges E = {01, 12, 23, 34, 04, 14, 13}.

Terms:

* **Neighbors**: Two vertices u and v are neighbors if they’re a edge (u, v) connecting between them.

E.g.: 0 and 1 are neighbors, 0 and 4 are neighbors

neighbors(0) = {1, 4}

* **Degree**: Degree(v) is the number of edges connected to v.

E.g.: degree(1) = 3, degree(2) = 2

* **Path**: Sequence of vertices connected by edges.

E.g.: 0 -> 1 -> 2 -> 3 is a path

* **Path length***:* Number of edges in a path.

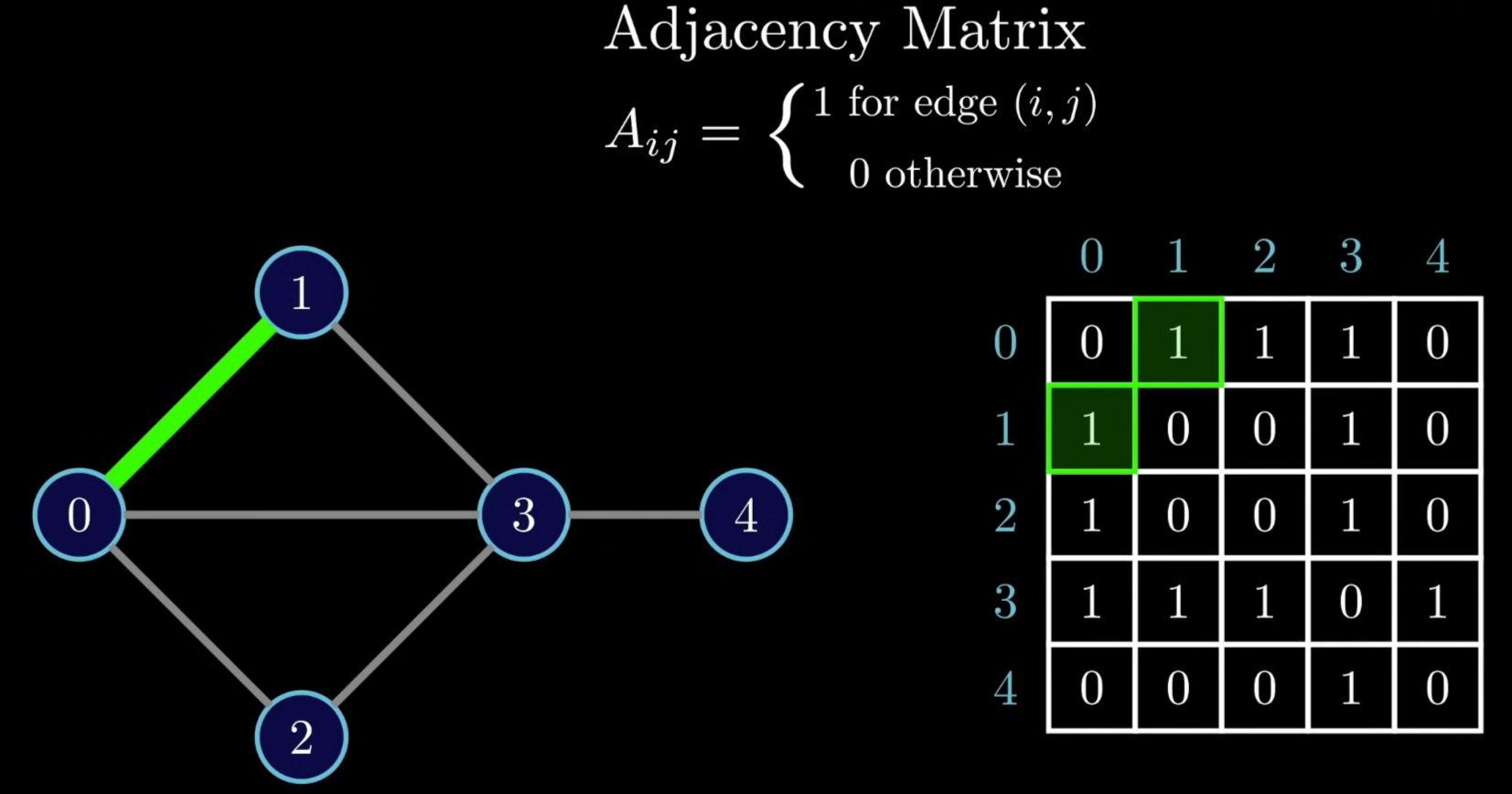
*E.*g.: 0 -> 1 -> 2 -> 3 has length of 3

* **Cycle**: Path that starts and ends at the same vertex

*E*.g.: 0 -> 1-> 4 -> 0 is a cycle

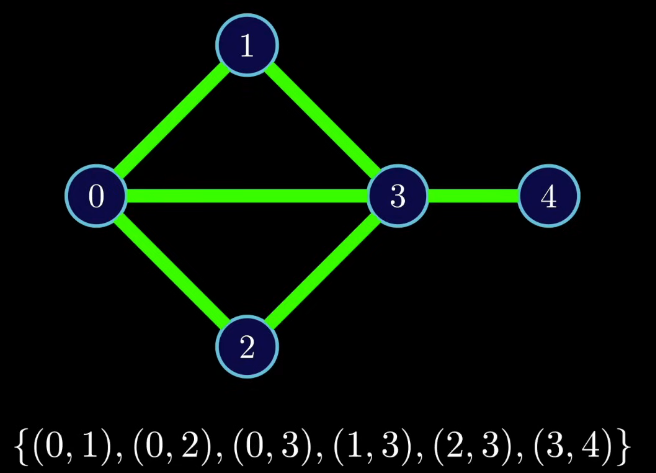
### Graph Representation

#### Adjacency Matrix



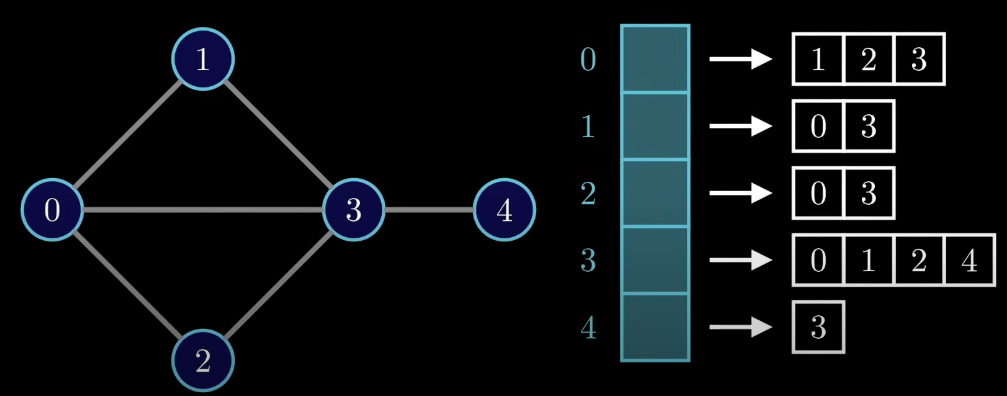
It’s complex, so not commonly used.

#### Edge Set



Commonly used.

#### Adjacency List



Commonly used, especially for space graph (= big graph).

### Traversal

#### Breadth First Search (BFS)

**Problem**

A BFS implementation puts each vertex of the graph into one of two categories:

* Visited
* Not visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

**Algorithm**

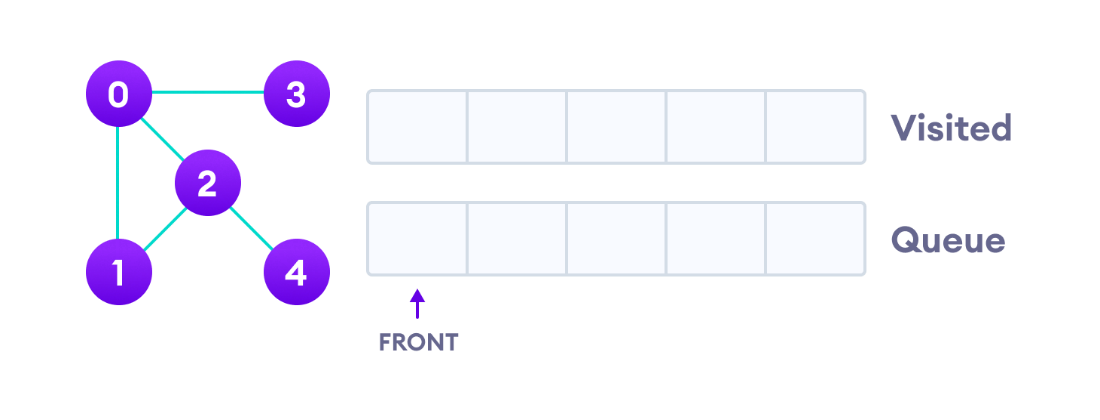
The algorithm works as follows:

1. Start by putting a starting vertex (any one of the graph's vertices) at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Find all of that vertex's adjacent nodes. And add those aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

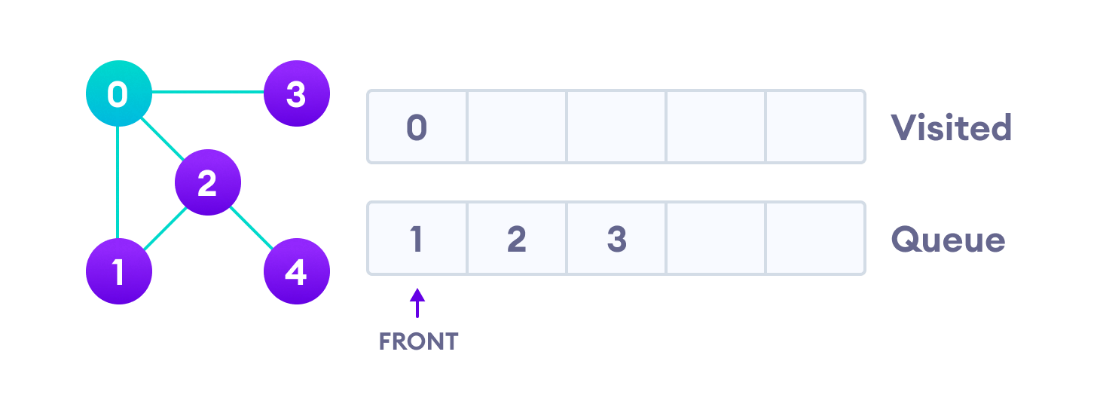
Note: The graph might have two different disconnected parts, so to make sure that we cover every vertex, we can also run BFS on every node.

**Example**

We create an undirected graph with 5 vertices:

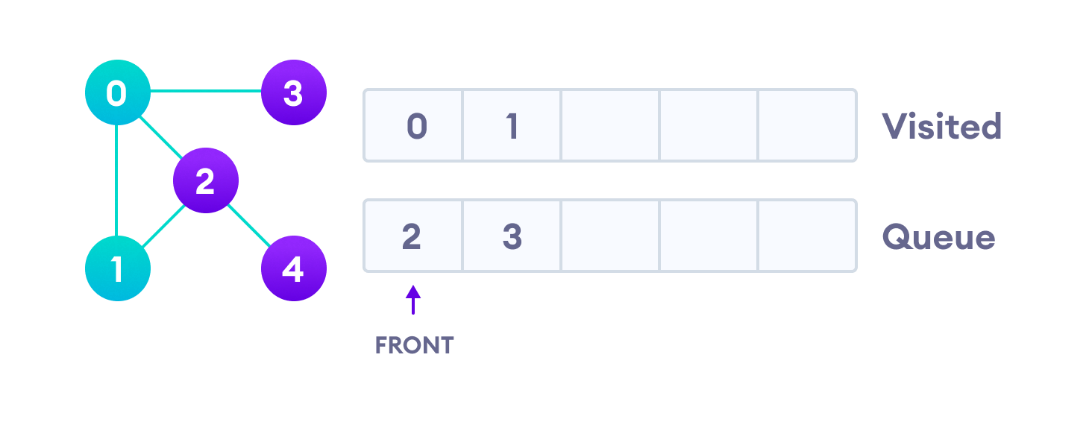


We choose vertex 0 as starting point. BFS puts it in the Visited list and puts all its adjacent vertices in the queue:

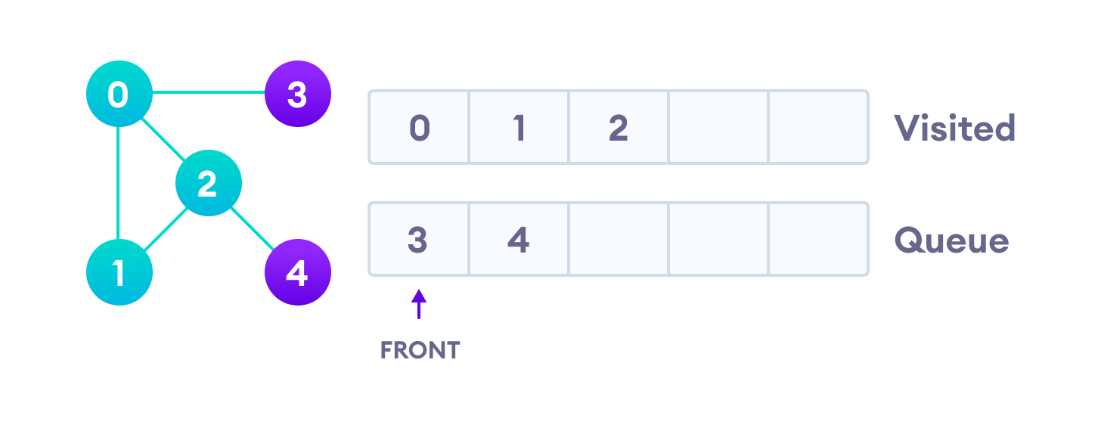


Next, we visit the element at the front of queue – which is 1, so we put 1 to the Visited list.

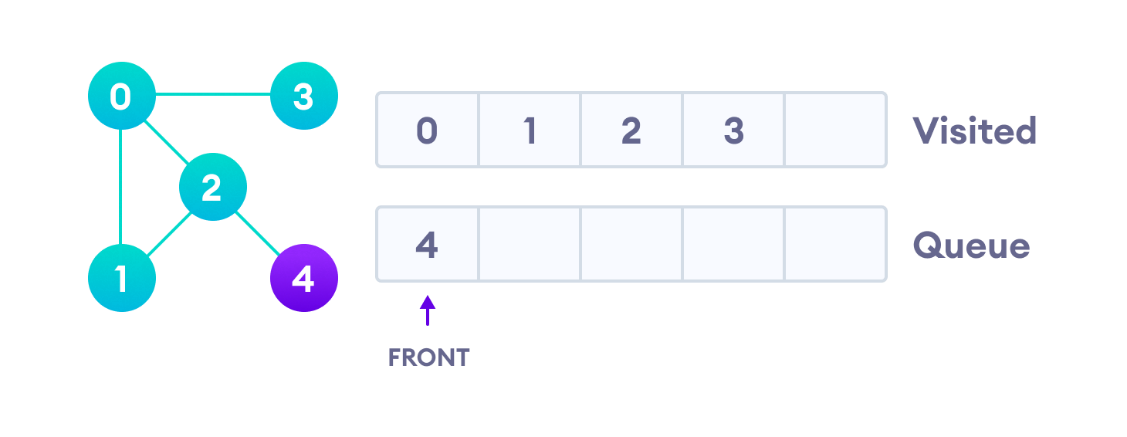
Then go to its adjacent nodes. Its adjacent vertices are 0 and 2. Because 0 has already been visited and 2 has already in the queue, we keep the rest of the queue unchanged:



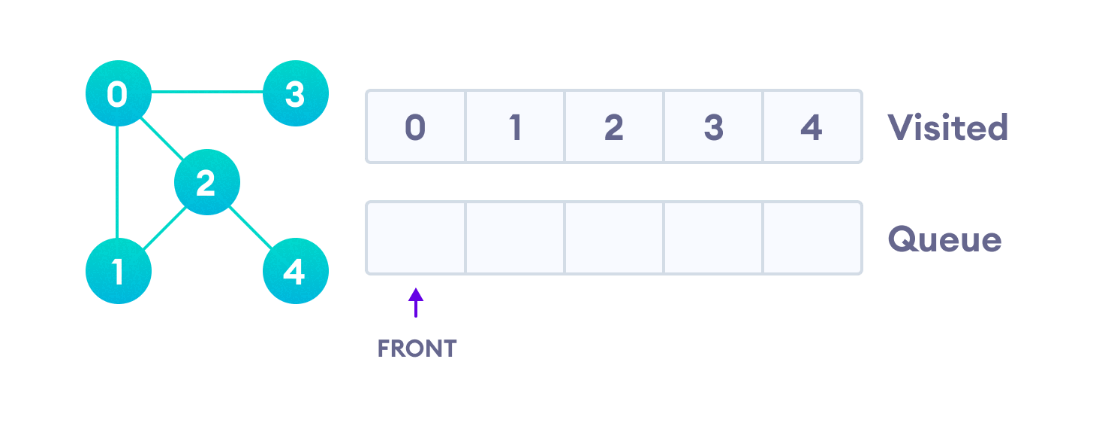
Next, we visit vertex 2, so we put 2 to the Visited list. This vertex has an unvisited adjacent vertex – which is 4, so we put 4 to the back of the queue:



Next, we visit vertex 3, so we put 3 to the Visited list. This vertex has no unvisited adjacent vertex, so we keep the rest of the queue unchanged:



Only 4 remains in the queue. We visit it:



Because the queue is empty, we have completed the BFS of the graph.

**Example**

#### Depth First Search (DFS)

**Problem**

A BFS implementation puts each vertex of the graph into one of two categories:

* Visited
* Not visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

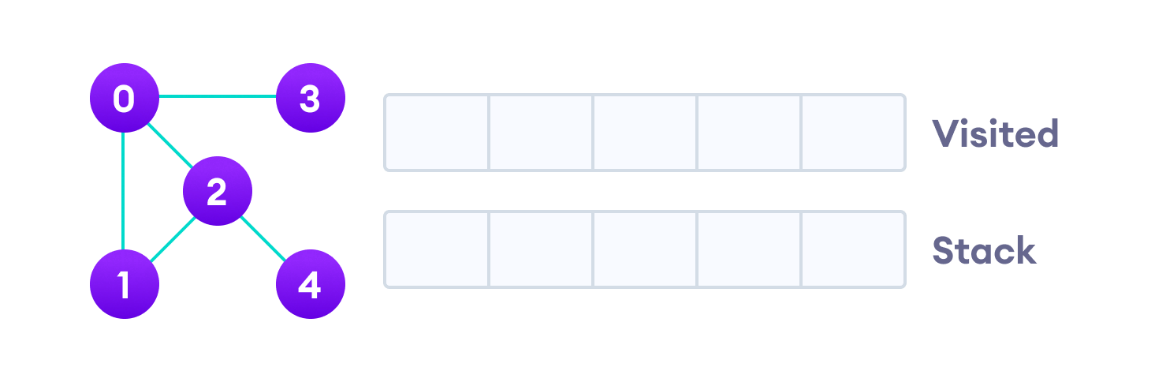
**Algorithm**

The algorithm works as follows:

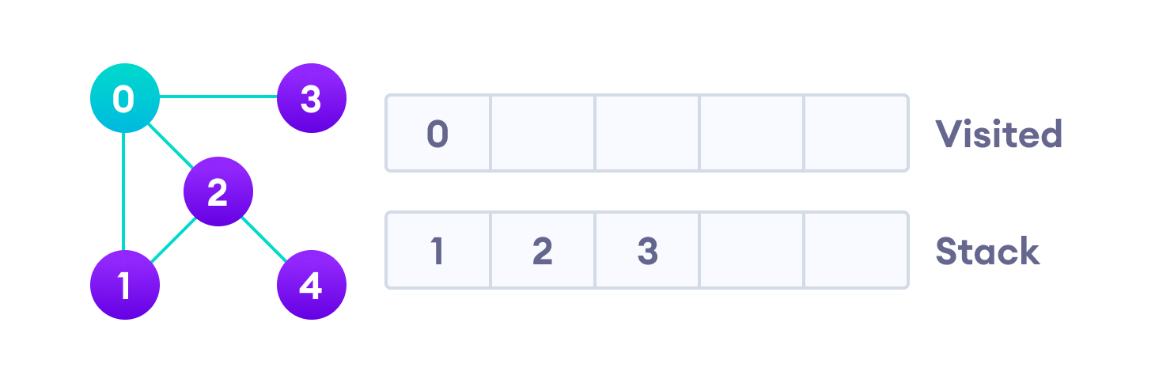
1. Start by putting a starting vertex (any one of the graph's vertices) at the back of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Find all of that vertex's adjacent nodes. And add those aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

**Example**

We create an undirected graph with 5 vertices:

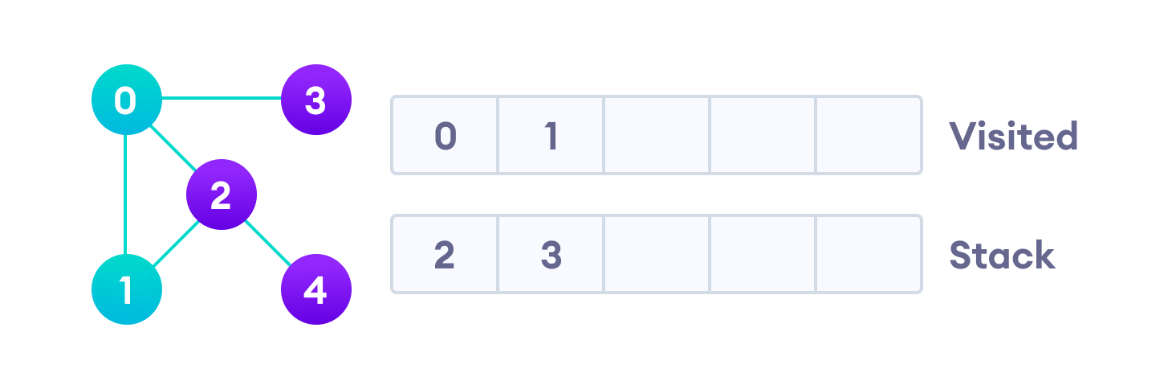


We choose vertex 0 as starting point. DFS puts it in the Visited list and puts all its adjacent vertices in the stack:

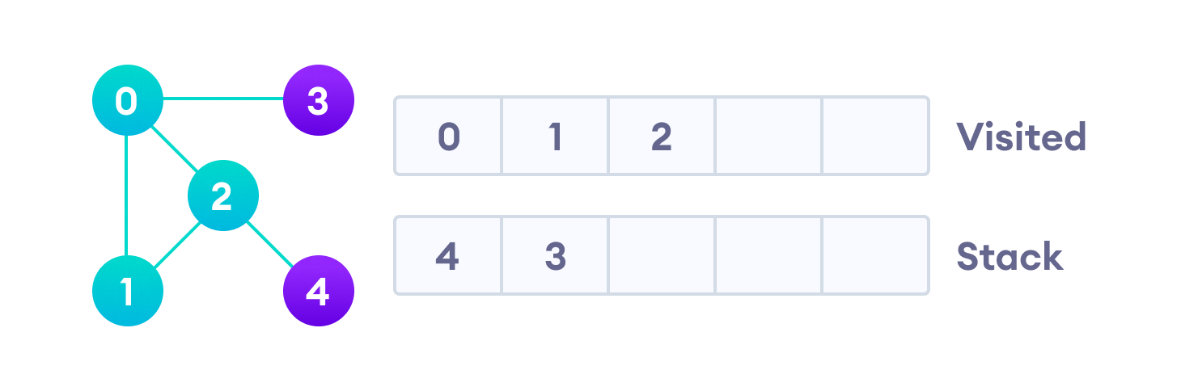


Next, we visit the element at the top of stack – which is 1, so we put 1 to the Visited list.

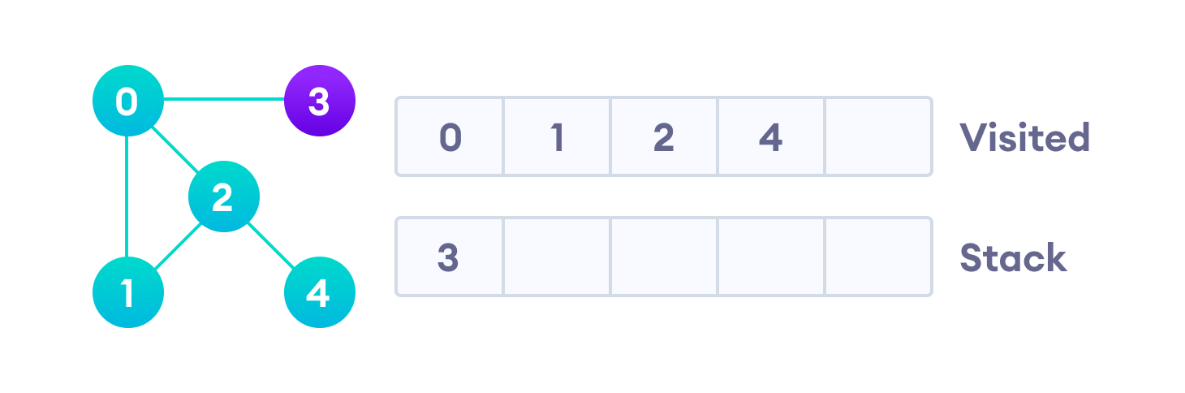
Then go to its adjacent nodes. Its adjacent vertices are 0 and 2. Because 0 has already been visited and 2 has already in the queue, we keep the rest of the v unchanged:



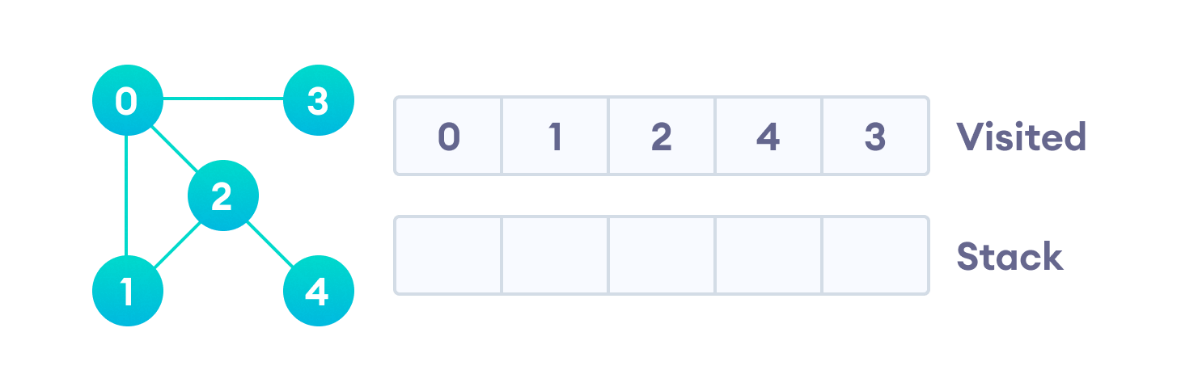
Next, we visit vertex 2, so we put 2 to the Visited list. This vertex has an unvisited adjacent vertex – which is 4, so we put 4 to the top of the stack:



Next, we visit vertex 4, so we put 4 to the Visited list. This vertex has no unvisited adjacent vertex, so we keep the rest of the stack unchanged:



Only 3 remains in the queue. We visit it:



Because the queue is empty, we have completed the BFS of the graph.

#### Uniform Cost Search (UCS)

### Operations & Complexities

#### Breadth First Search

Time complexity: O(V + E), where V is the number of nodes and E is the number of edges.

Sace complexity: O(V).

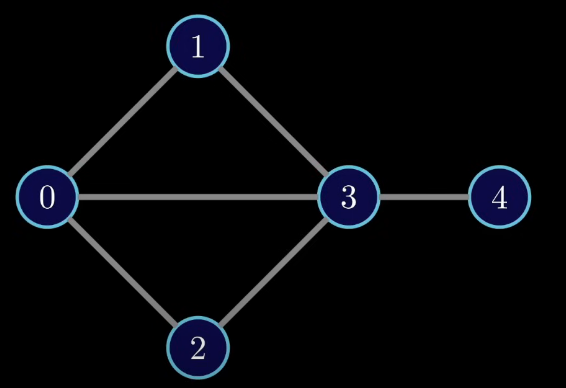
#### Depth First Search

Time complexity: O(V + E), where V is the number of nodes and E is the number of edges.

Sace complexity: O(V).

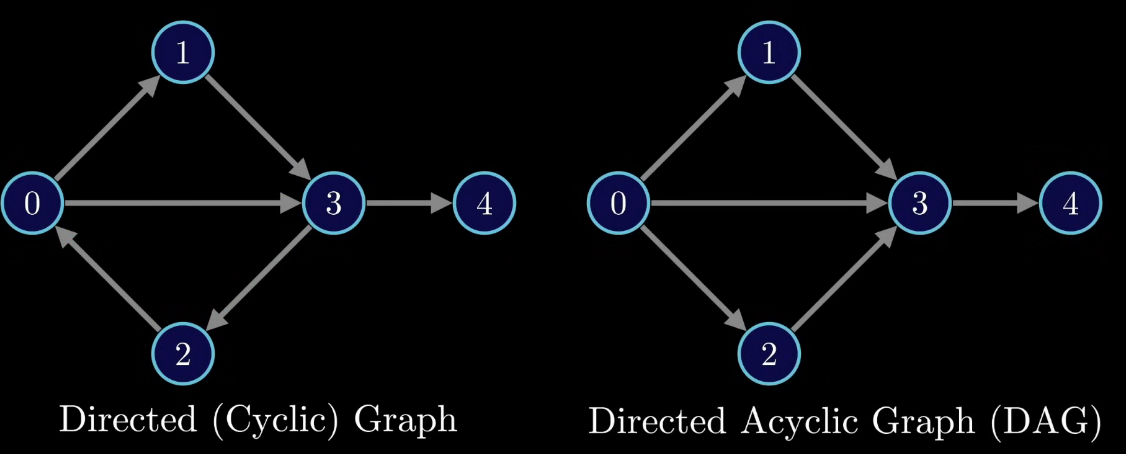
### Types of Graphs

#### Undirected Graph



Edge (u, v) implies edge (v, u)

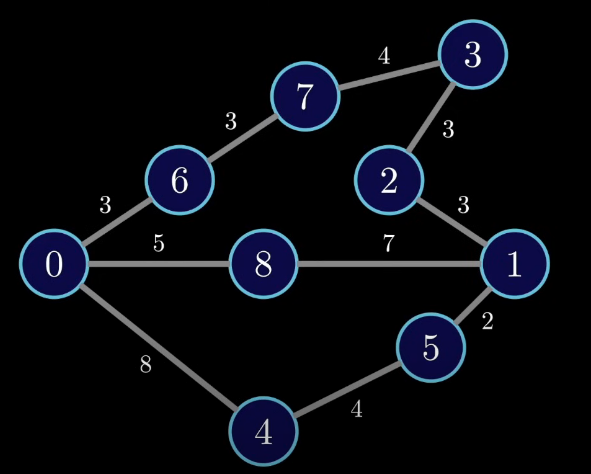
#### Directed Graph



Edges are unidirectional.

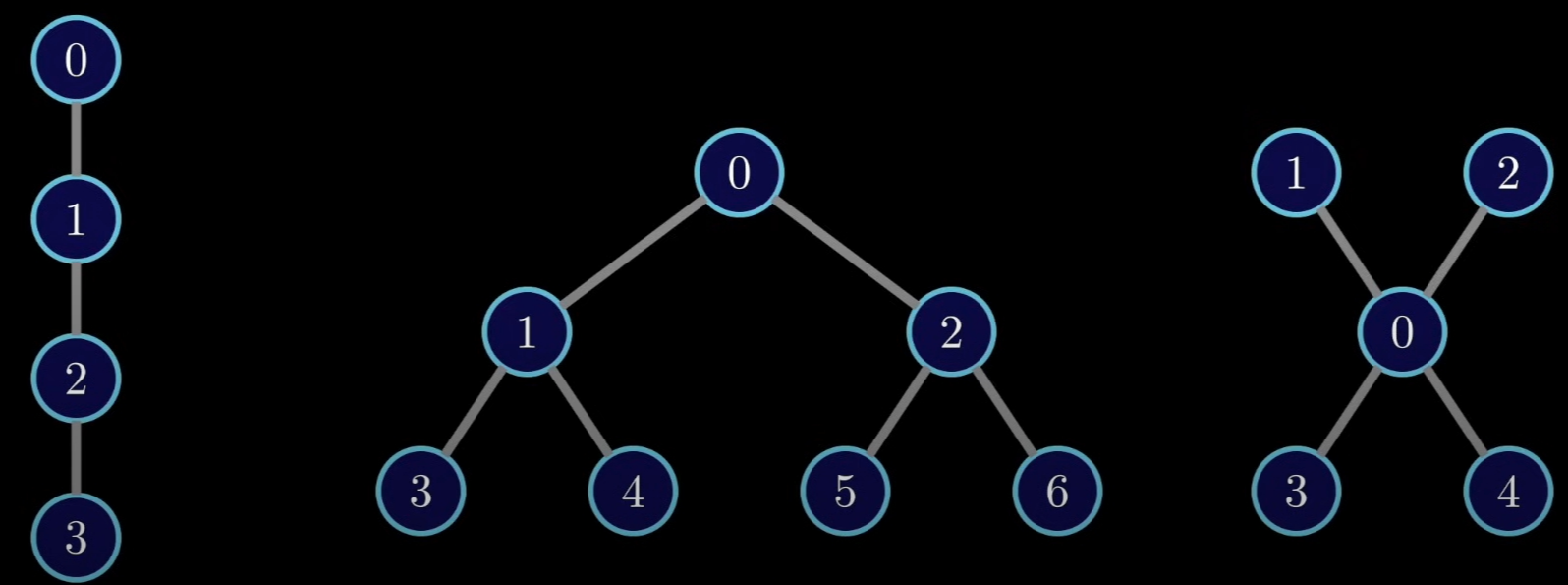
* **Directed Cyclic Graph**: A directed graph which contains at least one cycle.
* **Directed Acyclic Graph**: A directed graph which contains no cycle.

#### Weighted Graph



Each edge is not treated equally, but some edges have a larger weight than others.

#### Trees



Trees are a subset of graph. Check the [Tree session](#_Tree) for me details.

### Applications

Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network.

Graphs are used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, locale, etc.

Shortest path in maps

### Graph in C++ STL

C++ STL doesn’t provide graph library.

## Hashing Data Structure

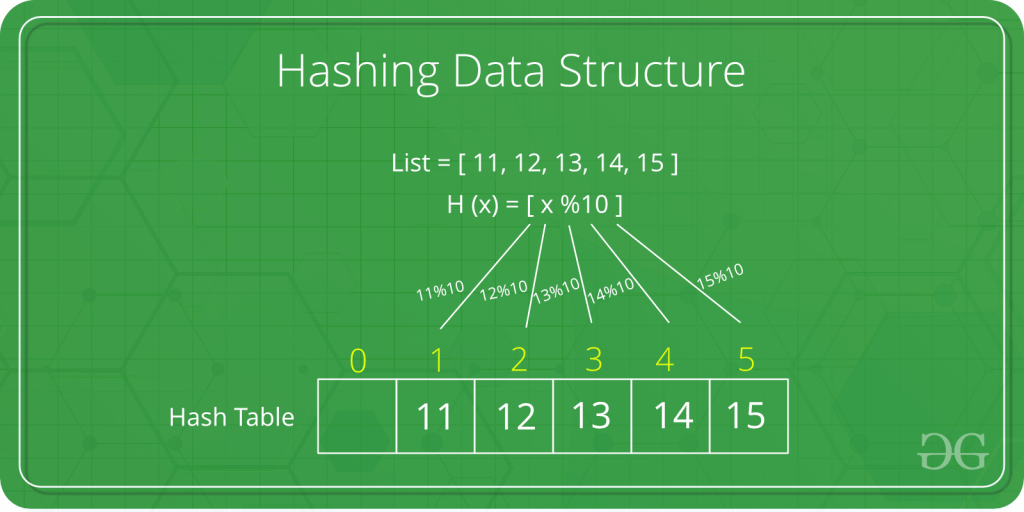
### Definition

Hashing is a popular technique for storing and retrieving data **as fast as possible**. That’s because hasing performs optimal searches.

**What?**

Hashing is a **technique of mapping keys and values into a hash table by using a hash function**. The efficiency of mapping depends on the efficiency of the hash function used.

Example: Hash function H(x) maps the value x at the index x % 10 in a list of [11,12,13,14,15]. The output hast table will store value at positions {1,2,3,4,5} respectively:



**Why?**

For example, in a Balanced Binary Search Tree, the time complexity for searching, inserting and deleting any element is O(logh). What if we want to do the same operations in a faster way? Here hashing comes into play.

In hashing, all the above operations can be performed in **O(1)**. Note thatin worst case, hashing remains O(n), but the average case only takes O(1).

### Operations & Complexities

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Meaning | Time Complexity | Space Complexity |
| HashTable | Create a new hash table | O(1) | O(1) |
| Delete | Delete a particular key-value pair from the hash table | O(1) | O(1) |
| Get | Search a key and return the value associated with that key | O(1) | O(1) |
| Put | Insert a new key-value pair | O(1) | O(1) |
| DeleteHashTable | Delete the hash table | O(1) | O(1) |

### Hashing Components

#### Hash Table

Hash table helps store data in a way that it's easy to find. This makes searching of an element very efficient.

#### Hash Function

Hash function **converts a given big number to a small integer value** **which is used as an *index* in hash table**.

In other words, a hash function is used to transform a given key into a specific slot index. Its main job is to map each and every possible key into a unique slot index. If every key is mapped into a unique slot index, then the hash function is known as a perfect hash function.

A **good hash function** should have following properties:

* Efficiently computable.
* Uniformly distribute the keys (each table position equally likely for each).
* Minimize collisions.
* Lave a low load factor (number of items in table divided by size of the table).

#### Collision Handling

Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a **newly inserted key maps to an already occupied slot in hash table is called collision**.

We must handle collisions. Following are some ways:

* **Chaining:** The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.

<https://www.geeksforgeeks.org/hashing-set-2-separate-chaining/>

* **Open Addressing:**In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we examine the table slots one by one until the desired element is found or it is clear that the element is not in the table.

<https://www.geeksforgeeks.org/hashing-set-3-open-addressing/>

### Applications

Design a system for storing employee records using phone numbers as keys.

We can think of using the following data structures to maintain information about different phone numbers:

* Array of phone numbers and records.
* Linked List of phone numbers and records.
* Balanced Binary Search Tree with phone numbers as keys.

We want the following queries to be performed efficiently:

* Insert a phone number and corresponding information.
* Search a phone number and fetch the information.
* Delete a phone number and related information.

But none of above data structures are really efficent. So we use hash table.

**Design for hash table:**

* A hash table stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.
* A hash function converts a big phone number to a small integer value which is used as index in the hash table. For phone numbers, a bad hash function is to take first three digits. A better function is consider last three digits.

### Hash Table in C++ STL

STL C++ doesn't provide library for hash table. But it provides equivalents:

* The equivalent of HashMap is std::unordered\_map
* The equivalent of HashSet is std::unordered\_set

# Algorithms

## Dynamic Programming (DB)

<https://www.geeksforgeeks.org/dynamic-programming/>

<https://www.youtube.com/watch?v=aPQY__2H3tE&list=PLpXOY-RxVRTM_-Lvss2ezy1lVl6VUrzW2&index=3&ab_channel=Reducible>

### What Is DP?

DP is an algorithmic technique that solves complex problems by breaking them down into simpler subproblems. By solving each subproblem only once and storing the results, it avoids redundant computations, leading to more efficient solutions for a wide range of problems.

### How Does DP Work?

* **Identify subproblems**: Divide the main problem into smaller, independent subproblems.
* **Store solutions**: Solve each subproblem and store the solution in a table or array.
* **Build up solutions**: Use the stored solutions to build up the solution to the main problem.
* **Avoid redundancy**: By storing solutions, DP ensures that each subproblem is solved only once, reducing computation time.

### Examples of DP

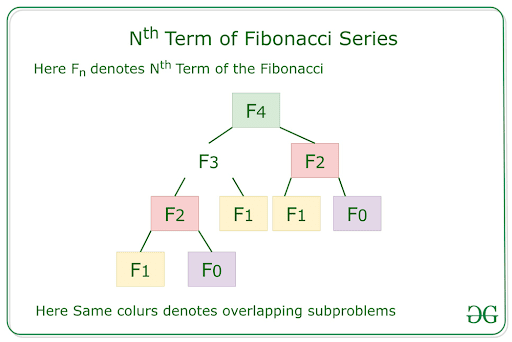
**Example 1:**Consider the problem of finding the Fibonacci sequence:

***Fibonacci sequence:****0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …*

**Brute Force Approach:**

To find the nth Fibonacci number using a brute force approach, you would simply add the **(n-1)th**and**(n-2)th** Fibonacci numbers. This would work, but it would be inefficient for large values of **n**, as it would require calculating all the previous Fibonacci numbers.

**Dynamic Programming Approach:**



Fibonacci series using DP:

* **Subproblems:**F(0), F(1), F(2), F(3), …
* **Store solutions:** Create a table to store the values of F(n) as they are calculated.
* **Build up Solutions:** For F(n), look up F(n-1) and F(n-2) in the table and add them.
* **Avoid redundancy:**The table ensures that each subproblem (e.g., F(2)) is solved only once.

By using DP, we can efficiently calculate the Fibonacci sequence without having to recompute subproblems.

### When to Use DP?

DP is an optimization technique used when solving problems that consists of the following characteristics:

1. **Optimal substructure**: Optimal substructure means that we combine the optimal results of subproblems to achieve the optimal result of the bigger problem.

*Example:*

*Consider the problem of finding the minimum cost path in a weighted graph from a source node to a destination node. We can break this problem down into smaller subproblems:*

*Find the minimum cost path from the source node to each intermediate node.*

*Find the minimum cost path from each intermediate node to the destination node.*

*The solution to the larger problem (finding the minimum cost path from the source node to the destination node) can be constructed from the solutions to these smaller subproblems.*

2. **Overlapping subproblems**: The same subproblems are solved repeatedly in different parts of the problem.

*Example:*

*Consider the problem of computing the Fibonacci series. To compute the Fibonacci number at index n, we need to compute the Fibonacci numbers at indices n-1 and n-2. This means that the subproblem of computing the Fibonacci number at index n-1 is used twice in the solution to the larger problem of computing the Fibonacci number at index n.*

### DP Algorithms

* Fibonacci Sequence: Calculates the nth Fibonacci number.
* Longest Common Subsequence (LCS): Finds the longest common subsequence between two strings.
* Shortest Path in a Graph: Finds the shortest path between two nodes in a graph.
* Knapsack Problem: Determines the maximum value of items that can be placed in a knapsack with a given capacity.
* Matrix Chain Multiplication: Optimizes the order of matrix multiplication to minimize the number of operations.

## Greedy

### What Is Greedy?

Greedy algorithms are a **class of algorithms that make locally optimal choices at each step with the hope of finding a global optimum solution**.

### It operates on the principle of “taking the best option now” without considering the long-term consequences.

### How Does Greedy Work?

* Define the problem: Clearly state the problem to be solved and the objective to be optimized.
* Identify the greedy choice: Determine the locally optimal choice at each step based on the current state.
* Make the greedy choice: Select the greedy choice and update the current state.
* Repeat: Continue making greedy choices until a solution is reached.

### Examples of Greedy

### When To Use Greedy?

Below are some applications of Greedy Algorithm:

* Assigning tasks to resources to minimize waiting time or maximize efficiency.
* Selecting the most valuable items to fit into a knapsack with limited capacity.
* Dividing an image into regions with similar characteristics.
* Reducing the size of data by removing redundant information.

Below are some disadvantages of greedy algorithm:

* Greedy algorithms may not always find the best possible solution.
* The order in which the elements are considered can significantly impact the outcome.
* Greedy algorithms focus on local optimizations and may miss better solutions that require considering a broader context.
* Greedy algorithms are not applicable to problems where the greedy choice does not lead to an optimal solution.

### Greedy Algorithms

* Fractional Knapsack: Optimizes the value of items that can be fractionally included in a knapsack with limited capacity.
* Dijkstra’s algorithm: Finds the shortest path from a source vertex to all other vertices in a weighted graph.
* Kruskal’s algorithm: Finds the minimum spanning tree of a weighted graph.
* Huffman coding: Compresses data by assigning shorter codes to more frequent symbols.

## Linked List

## Stack

### Checking Balancing of Brackets

#### Problem-1: Discuss how stacks can be used for checking balancing of brackets.

**Solution**

Stacks can be used to check whether the given expression has balanced brackets. This algorithm is very useful in compilers. Each time the parser reads one character at a time. If the character is an opening delimiter such as (, {, or [ - then it is written to the stack. When a closing delimiter is encountered like ), }, or ] - the stack is popped.

The opening and closing delimiters are then compared. If they match, the parsing of the string continues. If they do not match, the parser indicates that there is an error on the line.

A linear-time O(n) algorithm based on stack can be given as:

1. Create a stack.
2. while (end of input is not reached) {
3. If the character read is not a bracket to be balanced, ignore it.
4. Else if the character is an opening bracket like (, [, {, push it onto the stack.
5. Else if it is a closing symbol like ), ], }
6. If the stack is empty, report an error. Otherwise pop the stack.
   * 1. If the bracket popped is not the corresponding opening bracket, report an error.

}

1. At end of input, if the stack is not empty report an error.

Time Complexity: O(n). Space Complexity: O(n) [for stack]. Since we are scanning the input only once (using one loop).

## Searching

### Linear Search – O(n)

**Problem**

Given an array arr[] of n elements, write a function to search a given element x in arr[].

**Examples**

Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}; x = 110;

Output: 6 -> Element x is at index 6

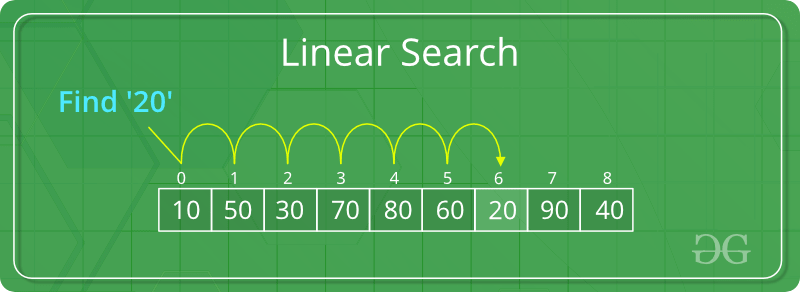
Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}; x = 175;

Output: -1 -> Element x is not in arr[].

**Algorithm**

Start from the leftmost element of arr[] and one by one compare x with each element of arr[]:

1. If x matches with an element, return the index.
2. If x doesn’t match with any of elements, return -1.



**Time Complexity**:O(n)

**Code**

<https://www.geeksforgeeks.org/linear-search/>

### Binary Search – O(1) or O(logn)

**Problem**

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

**Examples**

Input: arr[] = {10, 20, 30, 40, 50, 60}; x = 20;

Output: 1 -> Element x is at index 1

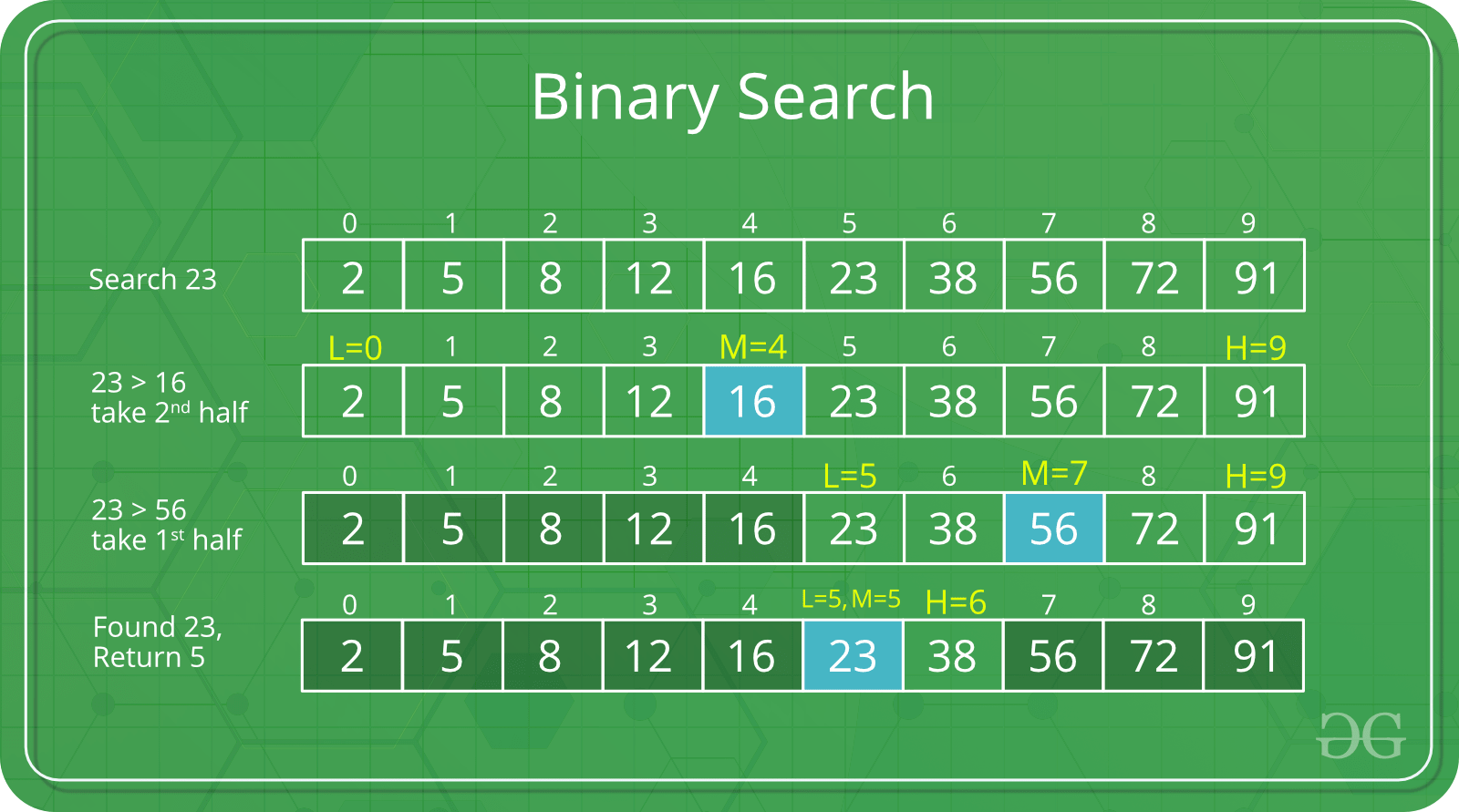
Input: arr[] = {10, 20, 30, 40, 50, 60}; x = 70;

Output: -1 -> Element x is not in arr[]

**Algorithm**

Compare x with the middle element in the array:

1. If x matches with middle element, return the mid index.
2. Else:
   1. If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
   2. Else (x is smaller) recur for the left half.



**Time Complexity**: O(1) if using iteration. Or O(logn) if using recursion.

**Code**

<https://www.geeksforgeeks.org/binary-search/>

### Ternary Search

### Jump Search – O(√n)

### Interpolation Search

### Exponential Search

### Fibonacci Search

### The Ubiquitous Binary Search

## Sorting

Visualization of the most famous sorting algorithms: <https://www.toptal.com/developers/sorting-algorithms>

### Selection Sort – O(n2)

**Problem**

Given an array arr[] of n elements, write a function to sort this array in an ascending order.

**Algorithm**

Repeatedly find the minimum element (considering ascending order) from unsorted part. Each time, the found element from the unsorted array is picked and moved to the sorted subarray (one by one, after each other).

The algorithm maintains two subarrays in a given array.

1. The subarray which is already sorted.
2. The remaining subarray which is unsorted.

Example: arr[] = { 64 25 12 11 }

// Find the minimum element in { 64 25 12 11 } and place it at the beginning of { 64 25 12 11 }:

**11** 25 12 64

// Find the minimum element in { 25 12 64 } and place it at the beginning of { 25 12 64 }:

11 **12** 25 64

// Find the minimum element in { 25 64 } place it at the beginning of { 25 64 }:

11 12 **25** 64

**Time Complexity**: O(n2) as there are two nested loops.

**Code**

<https://www.geeksforgeeks.org/selection-sort/>

#include <bits/stdc++.h>

using namespace std;

void selectionSort(int arr[], int n)

{

    int i, j, min\_idx;

    for (i = 0; i < n - 1; i++) {

        // Find the minimum element in unsorted array

        min\_idx = i;

        for (j = i + 1; j < n; j++) {

            if (arr[j] < arr[min\_idx])

                min\_idx = j;

        }

        // Swap the found minimum element with the first element

        if (min\_idx != i)

            swap(arr[min\_idx], arr[i]);

    }

}

void printArray(int arr[], int size)

{

    for (int i = 0; i < size; i++) {

        cout << arr[i] << " ";

    }

}

int main()

{

    int arr[] = { 64, 25, 12, 22, 11 };

    int n = sizeof(arr) / sizeof(arr[0]);

    selectionSort(arr, n);

    cout << "Sorted array: \n";

    printArray(arr, n);

    return 0;

}

### Bubble Sort – O(n2)

**Problem**

Given an array arr[] of n elements, write a function to sort this array in an ascending order.

**Algorithm**

Repeatedly swap adjacent elements in the array if they are in wrong order.

**Example**:

First pass:

{ **5 1** 4 2 8 } –> { **1 5** 4 2 8 }, compares the first two elements and swaps because 5 > 1.

{ 1 **5 4** 2 8 } –> { 1 **4 5** 2 8 }, swaps because 5 > 4

{ 1 4 **5 2** 8 } –> { 1 4 **2 5** 8 }, swaps because 5 > 2

{ 1 4 2 **5 8** } –> { 1 4 2 **5 8** }, because these elements are already in order, does not swap them.

Second pass:

{ **1 4** 2 5 8 } –> { **1 4** 2 5 8 }

{ 1 **4 2** 5 8 } –> { 1 **2 4** 5 8 }, swap because 4 > 2

{ 1 2 **4 5** 8 } –> { 1 2 **4 5** 8 }

{ 1 2 4 **5 8** } –> { 1 2 4 **5 8** }

Now, the array is already sorted, but our algorithm does not know if it is completed. It needs one more pass without any swap to know it is sorted.

Third pass:

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

**Time Complexity**: O(n2)

**Code**

<https://www.geeksforgeeks.org/bubble-sort/>

#include <bits/stdc++.h>

using namespace std;

void bubbleSort(int arr[], int n)

{

    int i, j;

    bool swapped;

    for (i = 0; i < n - 1; i++) {

        swapped = false;

        for (j = 0; j < n - i - 1; j++) {

            if (arr[j] > arr[j + 1]) {

                swap(arr[j], arr[j + 1]);

                swapped = true;

            }

        }

        if (swapped == false)

            break;

    }

}

void printArray(int arr[], int size)

{

    int i;

    for (i = 0; i < size; i++)

        cout << " " << arr[i];

}

int main()

{

    int arr[] = { 64, 34, 25, 12, 22, 11, 90 };

    int n = sizeof(arr) / sizeof(arr[0]);

    bubbleSort(arr, n);

    cout << "Sorted array: \n";

    printArray(arr, n);

    return 0;

}

### Quick Sort - O(nLogn)

**Problem**

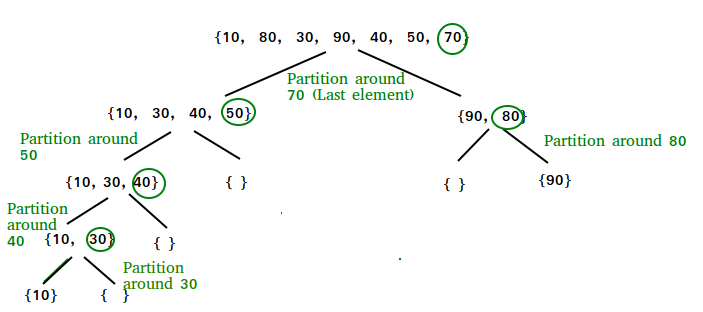
Given an array arr[] of n elements, write a function to sort this array in an ascending order.

**Algorithm**

Picks an element as pivot and partitions the array around that pivot. The pivot can be picked in different ways:

* Always pick first element as pivot.
* Always pick last element as pivot (illustration below).
* Pick a random element as pivot.
* Pick median as pivot.

The key process in quick sort is partition: put all smaller elements (than the pivot) before the pivot (if ascending order), and put all greater elements (than the pivot) after the pivot. All this should be done in linear time. Then place the pivot at its correct place.



For example:

arr[] = {10, 80, 40, 90, 30, 50, 70}

Index: 0 1 2 3 4 5 6

start = 0, end = 6, pivot = arr[end] = 70

Index of smaller element: i = -1

Traverse elements from j = start to end-1

j = 0 : Because arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 0, arr[] = {10, 80, 40, 90, 30, 50, 70} // No change as i and j are same

j = 1 : Because arr[j] > pivot, do nothing

j = 2 : Because arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 1, arr[] = {10, **40**, **80**, 90, 30, 50, 70} // We swapped 80 and 40

j = 3 : Because arr[j] > pivot, do nothing

j = 4 : Because arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2, arr[] = {10, 40, **30**, 90, **80**, 50, 70} // We swapped 80 and 30

j = 5 : Because arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i = 3, arr[] = {10, 40, 30, **50**, 80, **90**, 70} // We swapped 90 and 50

Loop ends because j = end-1.

We place pivot at correct position by swapping arr[i+1] and arr[end] (or pivot)

arr[] = {10, 40, 30, 50, **70**, 90, **80**} // We swapped 80 and 70

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

But we have not done sorting. For each sub-array {10, 40, 30, 50} and {90, 80}, we partition again (and might again and again …) until start > end.

The final result will be arr[] = {10, 30, 40, 50, 70, 80, 90}.

**Time Complexity**

Worst case: Θ(n2)

Best case: Θ(nLogn)

**Code**

<https://www.geeksforgeeks.org/quick-sort/>

#include <bits/stdc++.h>

using namespace std;

int partition(int arr[], int low, int high)

{

    // Choose the rightmost element as pivot

    int pivot = arr[high];

    // Index of smaller element and indicates the right position of pivot found so far

    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {

        // If current element is smaller than the pivot

        if (arr[j] < pivot) {

            i++;

            swap(arr[i], arr[j]);

        }

    }

    swap(arr[i + 1], pivot);

    return i + 1;

}

// low --> Starting index,

// high --> Ending index

void quickSort(int arr[], int low, int high)

{

    if (low < high) {

        // Partition the array and get the partitioning index

        int pi = partition(arr, low, high);

        // Recursively sort the sub-arrays

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

void printArray(int arr[], int size)

{

    int i;

    for (i = 0; i < size; i++)

        cout << " " << arr[i];

}

int main()

{

    int arr[] = { 10, 7, 8, 9, 1, 5 };

    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);

    cout << "Sorted array: " << endl;

    printArray(arr, n);

    return 0;

}

### Merge Sort - O(nLogn)

**Problem**

Given an array arr[] of n elements, write a function to sort this array in an ascending order.

**Algorithm**

Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

Example: arr[] = {38, 27, 43, 10}

Dividing step:

1st dividing: {38, 27, 43, 10} -> **{38, 27} and {43, 10}**

2nd dividing: {38, 27} -> **{38} and {27}**; {43, 10} -> **{43} and {10}**

Stop dividing because now no longer be divided more.

Merging step:

Sorted subarrays are merged together: {27, 38}, {10, 43}

Continuing mergeing until the sorted array is built from the smaller subarrays.: {10, 27, 38, 43}

**Time Complexity**: O(nLogn)

**Code**

<https://www.geeksforgeeks.org/merge-sort/>

#include <iostream>

using namespace std;

// Merges two subarrays of array[].

// First subarray is arr[begin..mid]

// Second subarray is arr[mid+1..end]

void merge(int array[], int left, int mid, int right)

{

    int subLeftArrLength = mid - left + 1;

    int subRightArrLength = right - mid;

    // Create temp arrays

    int\* leftArr = new int[subLeftArrLength];

    int\* rightArr = new int[subRightArrLength];

    // Copy data to temp arrays

    for (int i = 0; i < subLeftArrLength; i++)

        leftArr[i] = array[left + i];

    for (int j = 0; j < subRightArrLength; j++)

        rightArr[j] = array[mid + 1 + j];

    int subArr1Idx = 0;

    int subArr2Idx = 0;

    int mergedArrIdx = left;

    // Sort and merge the temp arrays back into array

    while (subArr1Idx < subLeftArrLength && subArr2Idx < subRightArrLength) {

        if (leftArr[subArr1Idx] <= rightArr[subArr2Idx]) {

            array[mergedArrIdx] = leftArr[subArr1Idx];

            subArr1Idx++;

        }

        else {

            array[mergedArrIdx] = rightArr[subArr2Idx];

            subArr2Idx++;

        }

        mergedArrIdx++;

    }

    // Copy the remaining elements of left[], if there are any

    while (subArr1Idx < subLeftArrLength) {

        array[mergedArrIdx] = leftArr[subArr1Idx];

        subArr1Idx++;

        mergedArrIdx++;

    }

    // Copy the remaining elements of right[], if there are any

    while (subArr2Idx < subRightArrLength) {

        array[mergedArrIdx] = rightArr[subArr2Idx];

        subArr2Idx++;

        mergedArrIdx++;

    }

    delete[] leftArr;

    delete[] rightArr;

}

void mergeSort(int array[], int left, int right)

{

    if (left >= right)

        return;

    // calculate the middle index of the array being divided

    //  (right - left): length of the subarray

    //  ((right - left) / 2): middle index within this subarray

    int mid = left + ((right - left) / 2);

    mergeSort(array, left, mid);

    mergeSort(array, mid + 1, right);

    merge(array, left, mid, right);

}

void printArray(int arr[], int size)

{

    for (int i = 0; i < size; i++)

        cout << arr[i] << " ";

}

int main()

{

    int arr[] = { 12, 11, 13, 5, 6, 7 };

    int n = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, n - 1);

    cout << "Sorted array is:\n";

    printArray(arr, n);

    return 0;

}

### Heap Sort - O(nLogn)

**Problem**

Given an array arr[] of n elements, write a function to sort this array in an ascending order.

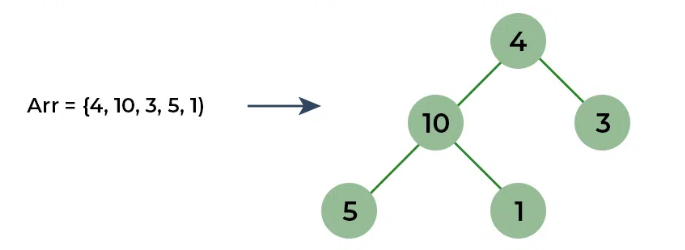
**Algorithm**

Steps:

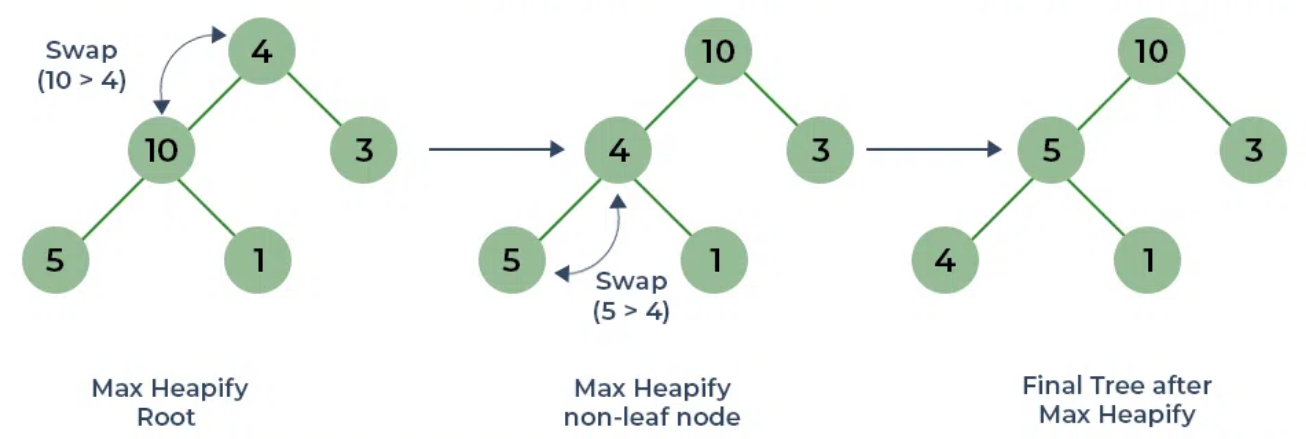
1. **Build heap**: Build a max heap from the input array.
   * Starting from the last non-leaf node, heapify each node in reverse order. Heapify is the process of adjusting the heap to maintain the max heap property.
   * Heapify compares the node with its children and swaps the node with the larger child if necessary. This process is recursively applied to the affected child until the max heap property is satisfied.
2. **Heapify and extract**: After building the max heap, the largest element (at the root) is in the correct position. We swap it with the last element of the heap (which is the last element of the array) and reduce the size of the heap by 1.
   * Swap the root with the last element of the heap.
   * Reduce the heap size by 1.
   * Heapify the root element to restore the max heap property.
3. **Repeat**: Repeat step 2 until the heap size is 1. This process moves the largest elements to the end of the array in ascending order.
4. **Sorted array**: The array is now sorted in ascending order.

Example:

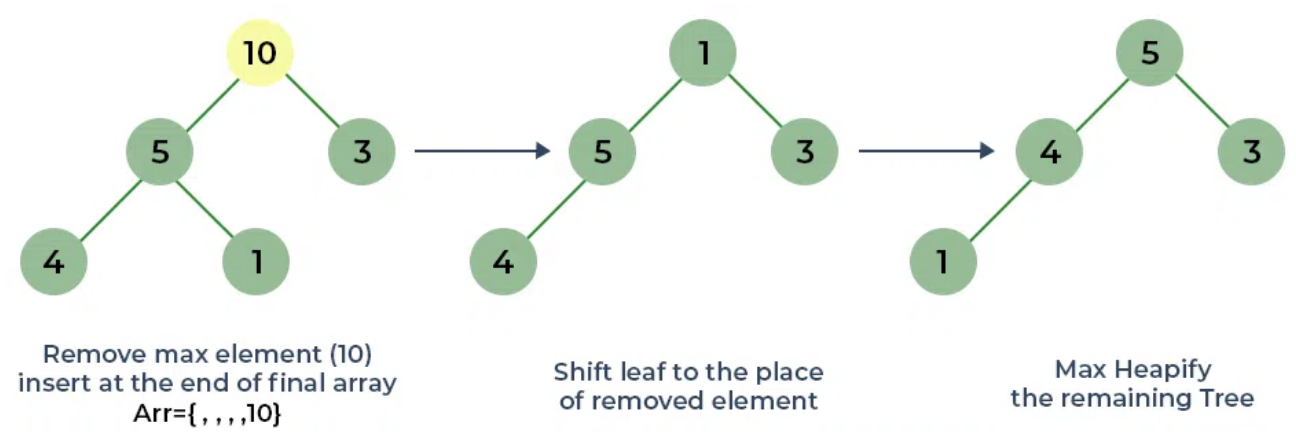
1. Initial array: {4, 10, 3, 5, 1}
2. Build a binary tree:



1. Build max heap:
   * Convert the array into a max heap: [10, 5, 3, 4, 1]



1. Heapify and extract:
   * Swap the root (10) with the last element (1): [**1**, 5, 3, 4, **10**]
   * Reduce the heap size by 1.
   * Heapify the root (1) to restore the max heap property: [5, 4, 3, 1, 10]



1. Heapify and extract:
   * Swap the root (5) with the last element (1): [1, 4, 3, 5, 10]
   * Reduce the heap size by 1.
   * Heapify the root (1) to restore the max heap property: [4, 1, 3, 5, 10]
2. Heapify and extract:
   * Swap the root (4) with the last element (3): [3, 1, 4, 5, 10]
   * Reduce the heap size by 1.
   * Heapify the root (3) to restore the max heap property: [1, 3, 4, 5, 10]
3. Heapify and extract:
   * Swap the root (1) with the last element (1): [1, 3, 4, 5, 10]
   * Reduce the heap size by 1.
   * Heapify the root (1) to restore the max heap property: [3, 1, 4, 5, 10]
4. Heapify and extract:
   * Swap the root (3) with the last element (1): [1, 1, 4, 5, 10]
   * Reduce the heap size by 1.
   * Heapify the root (1) to restore the max heap property: [1, 1, 4, 5, 10]
5. The heap size is now 1, and the array is sorted: [1, 1, 4, 5, 10]

The final sorted array is [1, 1, 4, 5, 10].

**Time Complexity**

**Code**

## Page Replacement Algorithms

In OS using paging for memory management, page replacement algorithms help **decide which page needed to be replaced** when there is a new page request, but there is not enough space in the main memory to allocate the new page.

Whenever a new page is referred and not present in memory, *page fault* occurs and OS replaces one of the existing pages with the new page.

Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

For details about paging technique, check Personal\Tutorials\Embedded Systems\Embedded System Tutorial.docx

### FIFO (First In First Out)

This is the simplest page replacement algorithm. In this algorithm, OS keeps track of all pages in the memory in a queue, the **oldest page is in the front of the queue, which will be selected for replaced**.

For example: Consider the page references 1, 3, 0, 3, 5, 6, 3 with 3 page frames:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 0 | 3 | 5 | 6 | 3 |
|  |  | 0 | 0 | 0 | 0 | 3 |
|  | 3 | 3 | 3 | 3 | 6 | 6 |
| 1 | 1 | 1 | 1 | 5 | 5 | 5 |
| Miss | Miss | Miss | Hit | Miss | Miss | Miss |

**Explanation:**

* Initially all slots are empty.
* When 1, 3, 0 came, they are allocated to the empty slots —> 3 Page Faults.
* When 3 comes, it is already in memory —> 0 Page Faults.
* When 5 comes, it is not available in  memory —> replaces the oldest page 1 —> 1 Page Fault.
* When 6 comes, it is not available in memory —> replaces the oldest page 3 —> 1 Page Fault.
* Finally, when 3 come, it is not available —> replaces the oldest page slot 0 —> 1 Page Fault.

**Algorithm:**

Input:

* Memory capacity (number of pages the memory can hold): N
* A string representing pages to be referred (stored in a list): L[i]

Output:

* Find number of page faults: PF
* Find the final queue which holding all pages after all operations: Q

Steps:

I - Start traversing the pages in a loop (the looping times = the number of requested pages).

1. If current page is present in list L, do nothing:

2. Else:

If list L has less pages than N:

a) Insert new page into list L.

b) Increase page fault PF by 1.

Else:

a) Remove the first page from queue Q as it was the first to be entered in the memory

b) Insert new page into list L.

d) Increase page fault PF by 1.

II - Return page fault PF.

III - Return queue Q.

Implementation (C++):

Personal\Tutorials\Data Structures - Algorithms\Code\PageReplacementAlgorithms

### LRU (Least Recently Used)

In this algorithm, **page which is least recently used will be replaced**.

For example, consider the page reference 7, 0, 1, 2, 0, 3, 0, 4, 2 with 4 page frames:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 |
|  |  |  | 2 | 2 | 2 | 2 | 2 | 2 |
|  |  | 1 | 1 | 1 | 1 | 1 | 4 | 4 |
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 7 | 7 | 7 | 7 | 3 | 3 | 3 | 3 |
| Miss | Miss | Miss | Miss | Hit | Miss | Hit | Miss | Hit |

**Explanation**:

* Initially all slots are empty
* When 7 0 1 2 are allocated to the empty slots —> 4 Page Faults.
* 0 is already their —> 0 Page Fault.
* When 3 came, it will take the place of 7 because it is least recently used —> 1 Page Fault.
* 0 is already in memory —> 0 Page Fault.
* 4 will takes place of 1 —> 1 Page Fault.
* 2 is already in memory —> 0 Page Fault.

**Algorithm:**

Input:

* Memory capacity (number of pages the memory can hold): N
* A string representing pages to be referred (stored in a list): L[i]

Output:

* Find number of page faults: PF
* Find the final list which holding all pages after all operations: Q

Steps:

I - Start traversing the pages in a loop (the looping times = the number of requested pages).

1. If current page is present in list L:

a) Increase page frequency by 1

2. Else:

If list L has less pages than N:

a) Insert new page into list L.

b) Increase page frequency by 1

c) Increase page fault PF by 1.

Else:

a) Find the page which is least recently used.

b) Remove this page.

c) Insert new page into list L.

d) Increase page fault PF by 1.

II - Return page fault PF.

III - Return list L.

Implementation (C++):

Personal\Tutorials\Data Structures - Algorithms\Code\PageReplacementAlgorithms

### LFU (Least Frequently Used)

In this algorithm, **page which is least recently used will be replaced**.

For example, consider the page reference 1, 2, 3, 4, 2, 1, 5 with 3 page frames:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 2 | 1 | 5 |
|  |  | 3 | 3 | 3 | 1 | 1 |
|  | 2 | 2 | 2 | 2 | 2 | 2 |
| 1 | 1 | 1 | 4 | 4 | 4 | 5 |
| Miss | Miss | Miss | Miss | Hit | Miss | Miss |

**Explanation**:

* Initially all slots are empty
* When 1 2 3 are allocated to the empty slots —> 3 Page Faults.
* When 4 cames —> replace page 1 --> 1 Page Fault.
* 2 is already in memory —> 0 Page Fault.
* When 1 cames —> replace page 3 (because compared to page 2, page 3 is the least frequently used) --> 1 Page Fault.
* When 5 cames —> replace page 4 (because compared to page 1 and 2, page 4 is the least frequently used) --> 1 Page Fault.

**Algorithm:**

Input:

* Memory capacity (number of pages the memory can hold): N
* A string representing pages to be referred (stored in a list): L[i]

Output:

* Find number of page faults: PF
* Find the final list which holding all pages after all operations: Q

Steps:

I - Start traversing the pages in a loop (the looping times = the number of requested pages).

1. If current page is present in list L:

a) Increase page frequency by 1

2. Else:

If list L has less pages than N:

a) Insert new page into list L.

b) Increase page frequency by 1.

c) Increase page fault PF by 1.

Else:

a) Find the page which is least frequently used.

b) Remove this page.

c) Insert new page into list L.

d) Increase page fault PF by 1.

II - Return page fault PF.

III - Return list L.

Implementation (C++):

Personal\Tutorials\Data Structures - Algorithms\Code\PageReplacementAlgorithms

### OPT (Optimal Page Replacement)

In this algorithm, pages are replaced which would **not be used for the longest duration of time in the future**. This is the best page replacement algorithm as it gives the least number of page faults.