


Big O

- $O(1)$ algorithms are most efficient as they scale really well because the execution time is independent of the input size
 - like searching a value in map using its key
- $O(n)$ means the execution time is linearly dependent on the input size
 - searching for an element in an array
 - we usually take the worst case scenario like if the element we are looking for is at the end of the array of length n then its $O(n)$
 - $O(n)$ can be $O(1)$ if the element we are looking for is at the 0th index but this is not guaranteed
- nested loop doesn't always imply $O(n^2)$
 - but $O(n^2)$ can have nested loops
- when searching a 1-D loop twice it is $O(2n)$
 - so doing it n times it is $O(n^2)$
- suppose we have a 1-D array of length n and first we traverse from 0th index to $(n-1)$ th index
 - then in the 2nd pass we skip the 0th element and start from 1st element. next pass from skip the 0th, 1st element and start from 2nd element and so on
 - so we will have a total of $n-1$ passes for an array of length n
 - if we draw the array one below the other for every pass and erase the block we skip, we will get ∇ shape
 - this has base and height $n-1$. We ignore 1 so the square has a $O(n^2)$ and ∇ shape has $O(n^2/2)$ but constant doesn't matter so its still $O(n^2)$
- if a 2-D matrix is not a square and has sides $n \times m$ then it will have $O(n \times m)$
- $O(\log n)$

- we keep shrinking the array in half so we will never have to traverse all the n elements and always less than n so it will be $O(\log n)$
- so $n=2^x$ where n is the length and x is the no of times will have to split in half so take log both sides to get $O(\log n)$
- used for binary trees
- very efficient for sorted data
- $O(n \log n)$ is more efficient than $O(n^2)$ but less than $O(n)$
- imagine a tree which has 3 branches and each branch has 3 more branches and so on and if the height of the tree is n then $O(3^n)$
- in case of recursion if the recursive function calls itself 2 times it is $O(2^n)$ as each call creates 2 more calls and each of the 2 create 2 more so it can be 2^0 , 2^1 or 2^2 calls and so on
 - so if a recursive function is called in a loop m times then it is $O(m^n)$
- $O(n!)$ is the worst
- efficiency decreases as gradient increases
 -  time vs input size
- array/vector/string
 - read is $O(1)$
 - insert/delete are $O(n)$
- singly linked list(forward_list)
 - read is $O(n)$
 - insert/delete at the start is $O(1)$
 - at the end is $O(n)$
 - anywhere else $O(n)$
- doubly lined list(list)
 - read is $O(n)$

- insert/delete at the start is $O(1)$
 - at the end is $O(1)$
 - anywhere else $O(n)$
- dequeue
 - read is $O(n)$
 - insert/delete front/end is $O(1)$
- priority queue
 - read max/min element is $O(1)$ as already sorted
 - insert and delete is $O(\log n)$ as sorting is required
- stack
 - push, pop and top are $O(1)$
- queue
 - same as stack
- set
 - read, insert and delete is $O(\log n)$ as it is implemented using trees
- multiset
 - read and insert are $O(\log n)$ and delete is $O(k \log n)$ where k is the number of duplicate entries
- map / multimap
 - read, insert and delete are $O(\log n)$
- unordered map
 - read, insert and delete are $O(1)$ on average, $O(n)$ in the worst case
- pairs
 - read is $O(1)$

- c++ STL is a collection of useful algorithms, data structures and functions related to them in form of template classes
- c++ STL has 4 things: containers, functions, algorithms and iterators
 - containers are the classes for the data structures
 - container functions are primitive operations performed on the containers like push, pop
 - algorithms are complex operations performed on the containers like search, sort
 - iterators are used to access containers in a simpler way
- to import all the containers and common header files like iostream etc, use `#include<bits/stdc++.h>`
- insert vs emplace
 - all containers might not have push operation but all have insert and emplace operations
 - both are same when using primitive data types
 - but when we use structs or objects emplace is more efficient
 - when we use insert it calls the constructor of the struct or object then it copies the struct or object and this copy is inserted
 - emplace just copies the object without constructing it

- Containers

1. array

- inside the array template class a primitive array is used and `.data()` is a function that returns that underlying primitive array
- an STL array can be indexed using overloaded `[]` operator or using `.at(x)` function
 - `.at(x)` function will throw an exception if the index is out of bounds but `[]` will give garbage value instead
- length provided during declaration has to be a constant and can never be a variable so you cant user input the length
 - these arrays can be passed by value to a function without any decay as the size can be found out by `.size()` function and the size is always known before compile time

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    array<int, 10> arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    cout << arr.at(4) << endl;    // 1
    cout << arr.front() << endl;  // 1
    cout << arr.back() << endl;   // 10 returns the last element of the array container
    cout << arr.size();           // 10 len of array

    cout << *arr.begin() << endl;    // 1 returns the address of the first element of the
    array container and is used for looping
    cout << *(arr.end() - 1) << endl; // 10
    cout << *arr.rbegin() << endl;   // 10
    cout << *(arr.rend() - 1) << endl; // 1
    cout << arr.empty() << endl;    // 0 checks if the array container is empty or not

    // cout<<arr.swap(anotherArrContainer);
```

```

    arr.fill(7); // array has all 10 elements equal to 7 and fill function is used to
    initialise

    return 0;
}

```

2. vector

- its a large array but not a linked list so when the array is filled then a new array is created and all the contents of the older array is copied to the new array

```

#include <bits/stdc++.h>
using namespace std;
int main()
{
    vector<int> v;

    v.push_back(1); // {1}
    v.push_back(2); // {1, 2}

    // emplace_back is faster than push_back
    v.emplace_back(3); // [1, 2, 3]
    v.emplace_back(4); // [1, 2, 3, 4]
    v.emplace_back(5); // [1, 2, 3, 4, 5]
    v.emplace_back(6); // [1, 2, 3, 4, 5, 6]

    v.erase(v.end() - 1); // to remove last element [1, 2, 3, 4, 5]
    v.erase(v.begin()); // removes 1st element [2, 3, 4, 5]
    // v.erase(startingIndexAddress,AdrOfLastElementToBeDeleted+1) can erase a range of
    elements if you provide

```

```

v.insert(v.begin(), 0); // [0, 2, 3, 4, 5]
// v.insert(address, howManyCopiesOf, ThisElement);
// to insert vector cpy into v: v.insert(Address, cpy.begin(), cpy.end());

cout << v.front() << " " << v.back() << endl; // 0 5

for (auto itr = v.begin(); itr != v.end(); itr++) // 0 2 3 4 5
    cout << *itr << " ";

cout << endl;

for (auto itr = v.rbegin(); itr != v.rend(); itr++) // 5 4 3 2 0
    cout << *itr << " ";

cout << endl;

v.assign({1, 2, 3, 4, 5});
for (auto x : v) // 1 2 3 4 5
    cout << x << " ";

cout << endl;

// v.capacity() is the max length of the vector including buffer capacity
// v.resize() changes the size and not the capacity but can change capacity if the size is
full
// v.reserve(1000) changes the capacity so that the vector doesn't resize till we push 1000
elements
// v.clear() resets the vector
// v.shrink_to_fit() releases unused memory
// v.at(), v.swap(), v.data(), v.empty(), v.size() is same as arrays

```

```

vector<pair<int, int>> vpair;
vpair.push_back({1, 2}); // need to use flower braces to specify a pair
vpair.emplace_back(3, 4); // automatically infers that it is a pair with emplace_back


vector<int> hundred(5, 100); // creates {100,100,100,100,100}
vector<int> bydefault(5);    // creates {0,0,0,0,0}

vector<int> temp(v); // creates a separate copy of v and names it temp

return 0;
}

```

3. list

- doubly linked list
-  doubly linked list

```

#include <bits/stdc++.h>
using namespace std;
int main()
{
    list<int> ls; // same as vector but also allows front operations
    ls.push_back(2);
    ls.emplace_back(4);
    ls.push_front(0);
    ls.emplace_front(-2);
    ls.pop_front();
    ls.pop_back();
    list<int> ls2;
}

```



```

    ls.splice(ls.begin(), ls2); // this will slice ls from ls.begin() and merge the sliced part
    with ls2
    ls.unique(); // will remove all duplicates from ls only if the duplicates are adjacent in
    positions so you might want to sort beforehand
    ls.remove(2); // will remove all occurrences of 2 from the ls
    ls.remove_if([] (int n) { // lambda function that removes all values grtr than 4 from the
    list
        return n>4;
    });
    ls.resize(2); // resizes the list to 2 so if there are more than 2 elements they are
    deleted otherwise the list is extended with values 0
    ls.sort(); // O(n log(n))
    return 0;
}

```

4. forward list

- same as list but singly linked list instead
- can traverse only in one direction

5. deque

- doubly ended queues are queues but elements can be pushed and popped from both ends
- kinda like vectors but contiguous memory allocations is not guaranteed but front operations are allowed along with back operations
 - implementation uses multiple arrays of fixed size such that the end of one array points to the start of another array
 - in vector push and pop are done from back but for deque they can be done from both front and back
 - imagine deque as parallel arrays of same size

- so if we have 10 parallel arrays of size 5 and we want 44th element then we can search $44\%5=4$ th elements of 44/5th array
- still random access read is $O(1)$
- expanding deque is cheaper than expanding vector as we will never have to copy all the elements like we might have to in vectors
- insertion and deletion at the front or back is $O(1)$ else it is $O(n)$

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    deque<int> dq;
    // dq.assign();
    // dq.at();
    // dq.front();
    // dq.back();
    // dq.begin();
    // dq.end();
    // dq.emplace_back();
    // dq.emplace_front();
    // dq.pop_back();
    // dq.push_front();

    // dq.clear();
    // dq.empty();
    // dq.erase();

    // dq.insert();
    // dq.resize();
    // dq.shrink_to_fit();
}
```

```
    // dq.size();  
    // dq.swap();  
    return 0;  
}
```

6. priority queue

- also called max heap/min heap
 - like a heap(FIFO) that stores the largest/smallest element on the top
- pop deletes the largest element

```
#include <bits/stdc++.h>  
using namespace std;  
int main()  
{  
    priority_queue<int> pq; // default is descending order  
    // priority_queue<int> pq;  
    // pq.emplace(); // emplace_back obviously as no front operations possible  
    // pq.empty();  
    // pq.pop(); // removes largest element  
    // pq.push();  
    // pq.size();  
    // pq.swap();  
    // pq.top(); // returns the topmost element of the heap which is the largest one  
    return 0;  
}
```

7. stack

- LIFO
- stack of washed plates
- same functions as priority_queue
- push adds new element at the top of the stack
- pop removes the topmost element
- top reads the topmost element
- useful for reversing without recursion
- insert and delete happens from the same end
- no iterator for a stack, you need to pass the stack object to another function by value which prints the top element and then pops in in a loop `while(!strack.empty())`

```
// reverse string using stack
#include <iostream>
#include <stack>
#include <cstring>
using namespace std;

void reverse(char str[], int len)
{
    stack<char> strack;
    for (int i = 0; i < len; i++)
        strack.push(str[i]);
    for (int i = 0; i < len; i++)
    {
        str[i] = strack.top();
        strack.pop();
    }
}
```

```

int main()
{
    char str[50];
    cout << "Enter String: ";
    cin >> str;
    reverse(str, strlen(str));
    cout << str;
    return 0;
}

```

- bracket balancing
 - in code editors the linter checks if an opening bracket in a expression is closed before the expression ends
 - this operation uses stack and is called check for balanced parenthesis
 - every opening bracket should have a closing pair so that it is balanced and expression is valid
 - saying number of opening brackets should be equal to number of closing ones is incomplete as `)a+b(` or `[()]` are invalid
 - so the rules are:
 - an opening bracket should have a closing one to its right and not its left
 - a bracket can be closed only if all the brackets opened inside it are already closed
 - last unclosed bracket should be closed first `[()()]`
 - scan from left to right
 - if opening bracket add it to the stack and if closing found then remove the topmost closing bracket type from the stack
 - at the end the stack should be empty to conclude the expression as valid

8. queue

- FIFO
- more used than stack irl
- used when there is a shared resource and it can only handle only one resource at a time like a printer or a processor
- no iterator for a queue, you need to pass the queue object to another function by value which prints the front/back element and then pops in in a loop `while(!q.empty())`

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    queue<int> q;
    // q.front();
    // q.back();
    // q.empty();
    // q.pop();
    // q.push();
    // q.size();
    // q.swap(anotherqueue);
    return 0;
}
```

9. set

- no duplicates so every element in a set is unique
- stored in ascending by default implemented using red-black/AVL tree

- `set<int, greater<>>` stores the elements in descending order
- but if we are providing objects/structs as template type then we need comparator as the set is implemented using a Binary Search Tree which needs a comparator for objects/structs sorting

```
#include <bits/stdc++.h>
using namespace std;

class student
{
public:
    string name;
    int roll;
    bool operator<(const student &rhs) const { return roll < rhs.roll; } // default
    ascending comparator
    bool operator>(const student &rhs) const { return roll > rhs.roll; } // default
    descending comparator
};

int main()
{
    cout << "Asc:" << endl;
    set<student> grade1 = {{ "A", 4}, {"B", 8}, {"C", 16}, {"D", 32}};
    for (auto x : grade1)
        cout << x.name << " " << x.roll << endl;
    cout << "Desc:" << endl;
    set<student, std::greater<>> grade2 = {{ "A", 4}, {"B", 8}, {"C", 16}, {"D", 32}};
    for (auto x : grade2)
        cout << x.name << " " << x.roll << endl;
}
```

```

#include <bits/stdc++.h>
using namespace std;
int main()
{
    set<int> s;

    // s.count(); // returns number of occurrences of an element in the set and makes sense
    only for multisets as for sets it is always 1

    // s.find(); // returns iterator pointing to the desired element and returns s.end() if
    not found

    // s.insert(); // element to be inserted is passed and it returns a pair of the
    iterator pointing to place where it might be inserted, bool if it was inserted or not

    // s.emplace();
    // s.begin();
    // s.end();
    // s.clear();
    // s.empty();
    // s.erase();
    // s.size();
    // s.swap();
    return 0;
}
...

```

10. `multiset`

- same as set with the same functions but allows duplicate entries and still are sorted
- `.equal_range(key)` used with `auto` returns an iterable container of values as the same key can have multiple values so we can use `auto` with `for` each loop to print all values of this container
 - prefer using `auto` as the return type is pair of 2 iterators where one is for the

keys **and** the other **for** values

- use **this** instead of map of key **and** vector value

11. **map**

- like arrays but the indices are called keys **and** keys can be of any data type
- stores key-value pairs **and** all keys must be unique **and** keys are stored in sorted order
- functions like set
- need to specify comparator a **template** type is a **struct/object** as it is implemented **using**

red-black/AVL tree

- ``.first`` gives the key **and** ``.second`` gives the value when **using** an iterator
- ``.someMap[key]`` returns value
- keys are immutable **and** values are **mutable**
- insertion **requires** passing of a pair as a parameter like ``.insert(make_pair(1, 'a'))``

where the type of pair is inferred automatically

- otherwise the type of the pair can be explicitly provided like ``.insert(pair<int,`

`char>(1, 'a'))``

12. **multimap**

- same as a map but allows duplicate keys but the key-value pairs must be unique
- hence ``.at()`` function **not** available

13. **unordered map**

- like map but keys are **not** stored in sorted order
- has some unique bucket functions

14. **pairs and nested pairs**

```cpp

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
 pair<int, int> p = {1, 2};
```

```
 cout << p.first << " " << p.second << endl;
```

```
 pair<pair<int, int>, int> np = {p, 3};
```

```
 cout << np.first.first << " " << np.first.second << " " << np.second << endl;
```

```

 pair<int, int> pairray[] = {{1, 2}, {3, 4}, {5, 6}, {7, 8}};
 for (int i = 0; i < sizeof(pairray) / 8; i++)
 cout << pairray[i].first << " " << pairray[i].second << endl;
 return 0;
 }
 ...

```

#### 15. <mark style="background: #BBFABBA6;">tree</mark>

- a vector of vector can be interpreted as a STL tree
  - every element in the parent vector is a child vector
  - the index of the parent array represents the parent node of the tree and the

associated vector stores the child node of the parent

```

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<vector<int>> tree;
    int edge, parentIndex, childValue;
    cin >> edge; // = len of the parent vector
    tree.resize(edge + 1);

    // to populate the tree
    for (int i = 0; i ≤ edge; ++i)
    {
        cout << "Enter Parent Index: ";
        cin >> parentIndex;
        cout << "Enter the value it points to: ";
        cin >> childValue;
        tree[parentIndex].push_back(childValue);
    }
}

```

```

        cout << endl;
    }

    int ctr = 0;
    for (const auto &p : tree)
    {
        cout << "\nParent Node " << ctr++ << ": points to children nodes ";
        for (const auto &c : p)
            cout << c << " ";
    }
    return 0;
}

```

```
/*
```

```
9
```

```
Enter Parent Index: 1
```

```
Enter the value it points to: 2
```

```
Enter Parent Index: 1
```

```
Enter the value it points to: 3
```

```
Enter Parent Index: 2
```

```
Enter the value it points to: 4
```

```
Enter Parent Index: 2
```

```
Enter the value it points to: 5
```

```
Enter Parent Index: 3
```

```
Enter the value it points to: 6
```

```
Enter Parent Index: 3
```

```
Enter the value it points to: 8
```

Enter Parent Index: 4
Enter the value it points to: 0

Enter Parent Index: 5
Enter the value it points to: 0

Enter Parent Index: 6
Enter the value it points to: 0

Enter Parent Index: 8
Enter the value it points to: 0

Parent Node 0: points to children nodes
Parent Node 1: points to children nodes 2 3
Parent Node 2: points to children nodes 4 5
Parent Node 3: points to children nodes 6 8
Parent Node 4: points to children nodes 0
Parent Node 5: points to children nodes 0
Parent Node 6: points to children nodes 0
Parent Node 7: points to children nodes
Parent Node 8: points to children nodes 0
Parent Node 9: points to children nodes
*/
...

- ![tree created]

(<https://raw.githubusercontent.com/JayaswalPrateek/MyCSnotesForME/main/Attachments/Screenshot%20from%202023-02-20%2010-17-53.png>)

- Iterators

- there are different iterators for different containers

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    // vector<int> v; by default creates a vector of len 16
    // vector<int> v(50); creates a vector of len 50
    vector<int> v = {1, 23, 456, 7890};
    v.push_back(56); // added 56 at the end after 7890
    v.push_back(69); // added 69 at the end after 56
    v.pop_back();    // last element is removed
    for (int x : v)
        cout << x << " ";
    cout << endl;
    for (vector<int>::iterator itr = v.begin(); itr != v.end(); itr++)
        cout << *itr << " "; // becoz itr is a ptr which uses ptr arithmetic
    return 0;
}
// we can replace occurrences of vector with list or forward_list or deque or set and not other
changes are needed
```

- using maps

```
#include <iostream>
#include <map>
using namespace std;
int main()
```

```

{
    map<int, string> m;
    m.insert(pair<int, string>(1, "john"));
    m.insert(pair<int, string>(2, "ravi"));
    m.insert(pair<int, string>(3, "khan"));

    map<int, string>::iterator itr;
    for (itr = m.begin(); itr != m.end(); itr++)
        cout << itr->first << " " << itr->second << endl;

    map<int, string>::iterator itr1;
    itr1 = m.find(2);
    cout << "value found: " << itr1->first << " " << itr1->second << endl;
    return 0;
}

```

CDT: Concrete Data Types

Array

```

#include <iostream>
using namespace std;
class concreteArray
{
    int *arrptr, size, len;
    void swap(int *x, int *y);

public:
    concreteArray(int size = 10)
    {

```

```

        this->size = size;
        len = 0;
        arrptr = new int[size];
    }
    ~concreteArray()
    {
        delete[] arrptr;
    }
    void Print(), Append(int x), Insert(int index, int x), Set(int index, int x) throw(string),
ReverseMethodOne(), ReverseMethodTwo(), InsertSort(int x);
    int Delete(int index) throw(string), LinearSearch(int key), BinarySearch(int key), Get(int index),
Min(), Max(), Sum();
    float Avg();
    bool isSorted();
};

void concreteArray::Print()
{
    if (len == 0)
    {
        cout << "Array is empty";
        return;
    }
    cout << "[ ";
    for (int i = 0; i < len; i++)
        cout << arrptr[i] << " ";
    cout << "]" << endl;
}

void concreteArray::Append(int x)
{

```

```
    if (len == size)
    {
        cout << "Array is full, cannot append" << endl;
        return;
    }
    arrptr[len] = x;
    len++;
}
```

```
void concreteArray::Insert(int index, int x)
{
    if (len == size)
    {
        cout << "Array is full, cannot insert" << endl;
        return;
    }
    else if (index < 0 || index > len)
    {
        cout << "Invalid index" << endl;
        return;
    }
    for (int i = len; i > index; i++)
        arrptr[i] = arrptr[i - 1];
    arrptr[index] = x;
    len++;
}
```

```
int concreteArray::Delete(int index) throw(string)
{
    if (index < 0 || index > len)
    {
```



```

        cout << "Invalid index" << endl;
        throw string("badIndex");
    }
    int deletedValue = arrptr[index];
    for (int i = index; i < len - 1; i++)
        arrptr[i] = arrptr[i + 1];
    len--;
    return deletedValue;
}

void concreteArray::swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

int concreteArray::LinearSearch(int key)
{
    for (int i = 0; i < len; i++)
        if (arrptr[i] == key)
        {
            if (i != 0)
                swap(&arrptr[i], &arrptr[0]);
            return i;
        }
    return -1;
}

int concreteArray::BinarySearch(int key)
{

```

```

int l = 0, h = len - 1;
while (l ≤ h)
{
    int mid = (l + h) / 2;

    if (key == arrptr[mid])
        return mid;
    else if (key < arrptr[mid])
        h = mid - 1;
    else
        l = mid + 1;
}
return -1;
}

int concreteArray::Get(int index) { return (index < 0 || index > len) ? -1 : arrptr[index]; }

void concreteArray::Set(int index, int x) throw(string)
{
    if (index < 0 || index > len)
    {
        cout << "Invalid index" << endl;
        throw string("badIndex");
    }
    arrptr[index] = x;
}

int concreteArray::Max()
{
    int max = arrptr[0];
    for (int i = 1; i < len; i++)

```

```

        max = arrptr[i] > max ? arrptr[i] : max;
    }
    return max;
}

int concreteArray::Min()
{
    int min = arrptr[0];
    for (int i = 1; i < len; i++)
        min = arrptr[i] < min ? arrptr[i] : min;
    return min;
}

int concreteArray::Sum()
{
    int sum = arrptr[0];
    for (int i = 1; i < len; i++)
        sum += arrptr[i];
    return sum;
}

float concreteArray::Avg() { return (float)(Sum() / len); }

void concreteArray::ReverseMethodOne()
{
    int *temparrptr = new int[size];
    for (int i = 0; i < len; i++)
        temparrptr[len - 1 - i] = arrptr[i];
    delete[] arrptr;
    arrptr = temparrptr;
    temparrptr = nullptr;
}

```

```

void concreteArray::ReverseMethodTwo()
{
    for (int i = 0; i < len; i++)
        swap(&arrptr[i], &arrptr[len - 1 - i]);
}

void concreteArray::InsertSort(int x)
{
    if (len == size)
    {
        cout << "Array is full, cannot insert" << endl;
        return;
    }
    int i;
    for (i = len - 1; i ≥ 0 && arrptr[i] > x; i--)
        arrptr[i + 1] = arrptr[i];
    arrptr[i + 1] = x;
    len++;
}

bool concreteArray::isSorted()
{
    for (int i = 0; i < len - 1; i++)
        if (arrptr[i] > arrptr[i + 1])
            return false;
    return true;
}

```

```

#include <iostream>
#include <climits>
using namespace std;

class LinkedList
{
    struct node
    {
        int data = 0;
        node *next = nullptr;
    } *head = nullptr;
    struct minmax
    {
        int min = 0, max = 0;
        minmax() = default; // default constructor, generated automatically but
not when there is a parametrized constructor declared
        minmax(int x, int y) : min(x), max(y) {} // parametrized constructor for the struct
    };
    int len = 0;
    minmax findMinMax(node *p);
    void dupliDeleter();

public:
    explicit LinkedList(int n = 5) // explicit can be used with any constructor that takes a single
argument to avoid unexpected behaviour
    {
        if (n < 1)
        {
            cout << "Length of Linked List cannot be smaller than 1";
            exit(-1);
        }
    }

```

```

        len = n;
        cout << "insert at beginning of the Linked List: elements that are filled first get pushed
towards the end" << endl;
        for (int i = len; i > 0; i--)
        {
            cout << "Enter element " << i << ": ";
            node *tmp = new node;
            cin >> tmp->data;
            tmp->next = head; // tmp->next now points at the 1st node of the linked list while the
head is also pointing at it
            head = tmp;      // head now points at the new node tmp and tmp->next was already
pointing to the node that was previously the first node
        }
        cout << "\nThe constructed Linked List of length " << len << " is:" << endl;
        printLinkedList();
    }
    LinkedList(const int arr[], int n)
    {
        len = n;
        cout << "creating Linked List from array elements" << endl;
        for (int i = n - 1; i ≥ 0; i--)
        {
            node *tmp = new node;
            tmp->data = arr[i];
            tmp->next = head;
            head = tmp;
        }
        cout << "\nThe constructed Linked List of length " << len << " is:" << endl;
        printLinkedList();
    }
    ~LinkedList()

```

```

{
    node *tmp=head;
    while (head != nullptr)
    {
        head = head->next;
        delete tmp;
        tmp = head;
    }
}

void printLinkedList(), recursivePrintLinkedList(node *p), insertDataAt(), deleteDataAt(),
printMinMax(), swappyLinearSearch(), amIsorted(), reverseByLinkFlipBySlidingPointers();
int recursiveAdd(node *p);
node *getHead() { return head; }
};

void LinkedList::printLinkedList()
{
    // cannot iterate directly using head otherwise we will lose its original value while incrementing
    when traversing Linked List
    cout << "The Head points to [" << head << "] and" << endl;
    int ctr = 1;
    for (node *itr = head; itr != nullptr; itr = itr->next)
        cout << "Node Number " << ctr++ << ": This is [" << itr << "]\t storing " << itr->data << "\t"
and the next address is 👉 [" << itr->next << "]" << endl;
}

void LinkedList::recursivePrintLinkedList(node *p)
{
    if (p != nullptr)
    {

```

```

        static int ctr = 1;
        cout << "Node Number " << ctr++ << ": This is [" << p << "]\t storing " << p->data << "\t and
the next address is 🖱️ [" << p->next << "]" << endl;
        recursivePrintLinkedList(p->next);
    }
}

void LinkedList::insertDataAt()
{
    // to insert a new node at the position pos we create a tmp node that is to be inserted
    // and then make the next of pos-1 th node point to it and store (pos-1)->next in tmp->next
    int pos;
    do
    {
        cout << "\nEnter Insertion point: ";
        cin >> pos;
    } while (pos < 0 || pos > len);
    cout << "Enter Value to be inserted at Node Number " << pos << ": ";
    node *tmp = new node;
    cin >> tmp->data;
    if (pos == 1)
    {
        tmp->next = head;
        head = tmp;
    }
    else
    {
        // lets go to the pos-1 th node
        node *itr = head;
        for (int i = 0; i < pos - 2; i++)
            itr = itr->next;
    }
}

```



```

        tmp->next = itr->next;
        itr->next = tmp;
    }
    len++;
    cout << "Added " << tmp->data << " at Node Number " << pos << " pushing the nodes after it 1 place
ahead\n\n";
    printLinkedList();
}

void LinkedList::deleteDataAt()
{
    // to delete an existing node you need to link the node before and after it by making the next of
    node before it point to the node after it
    // remember to free the deleted node at the end else it will be a memory leak
    int pos;
    do
    {
        cout << "\nEnter Node Number to be deleted: ";
        cin >> pos;
    } while (pos < 0 || pos > len);
    node *before = head;
    if (pos == 1)
    {
        head = before->next;
        delete before;
        return;
    }
    for (int i = 0; i < pos - 2; i++)
        before = before->next;
    node *after = before->next->next;
    delete before->next;

```

```

    before->next = after;
    cout << "Deleted Node Number " << pos << " pulling the nodes after it 1 place behind\n\n";
    printLinkedList();
}

int LinkedList::recursiveAdd(node *p)
{
    if (p == NULL)
        return 0;
    return recursiveAdd(p->next) + p->data;
}

LinkedList::minmax LinkedList::findMinMax(node *p)
{
    if (p == 0)
        return {INT_MAX, INT_MIN}; // parametrized constructor handles it
    minmax mm = findMinMax(p->next);
    return {mm.min > p->data ? p->data : mm.min, mm.max < p->data ? p->data : mm.max};
}

void LinkedList::printMinMax()
{
    minmax final = findMinMax(head);
    cout << "\nThe smallest value is " << final.min << " and the largest value is " << final.max <<
endl;
}

void LinkedList::swappyLinearSearch()
{
    /**
     * a swappy linear search is a variation of linear search
     * where if we find the key in a node we move the node to the start

```

```

* so that the next time searching will be quicker
*
* we have to iterators p and q such that q is always one node behind p
* when we find the key at p we link q node→next to p node→next
* and copy head into p node→next and make head point to p node now
*/
cout << "\nEnter Query: ";
int key;
cin >> key;
node *p = head, *q = nullptr;
while (p != nullptr)
{
    if (key == p->data)
    {
        q->next = p->next;
        p->next = head;
        head = p;
        cout << "Found! Pulled to head" << endl;
        printLinkedList();
        return;
    }
    q = p;
    p = p->next;
}
cout << "Not Found" << endl;
}

void LinkedList::amIsorted()
{
    bool flag = true;
    int compare = INT_MIN;

```

```

for (node *itr = head; itr != nullptr; itr = itr->next)
{
    if (itr->data < compare)
    {
        flag = false;
        break;
    }
    compare = itr->data;
}
if (flag)
{
    cout << "\nLinked List is sorted, Lets check for duplicates" << endl;
    dupliDeleter();
}
else
    cout << "\nLinked List is not sorted" << endl;
}

void LinkedList::dupliDeleter()
{
    bool flag = false;
    node *p = head, *q = head->next;
    while (q != nullptr)
    {
        if (p->data != q->data)
        {
            p = q;
            q = q->next;
            continue;
        } // below lines are executed when p->data == q->data
        p->next = q->next;
        delete q;
    }
}

```

```

        flag = true;
        q = p->next;
    }
    if (flag)
    {
        cout << "Deleted found duplicates" << endl;
        printLinkedList();
    }
    else
        cout << "No Duplicates were found" << endl;
}

void LinkedList::reverseByLinkFlipBySlidingPointers()
{
    node *p = head, *q = nullptr, *r = nullptr;
    while (p != nullptr)
    {
        r = q;
        q = p;
        p = p->next;
        q->next = r;
    }
    head = q;
    cout << "\nLinked List stored in reverse" << endl;
    printLinkedList();
}

int main()
{
    // int arr[] = {4, 8, 16, 32, 64};
    // LinkedList linkls(arr, 5);

```

```

LinkedList linkls;
linkls.insertDataAt();
linkls.deleteDataAt();
cout << "\nPrinting the Linked List recursively" << endl;
linkls.recursivePrintLinkedList(linkls.getHead());
cout << "\nThe sum of all elemets is " << linkls.recursiveAdd(linkls.getHead()) << endl;
linkls.printMinMax();
linkls.swappyLinearSearch();
linkls.amIsorted();
linkls.reverseByLinkFlipBySlidingPointers();
return 0;
}

```

Circular Linked list

- the last node of the linked list points to the head of the linked list instead of `nullptr`
- an empty linked list can never be circular as it needs at least one node and a head so that the head can point to node and node can point to the head
 - just like how a bidirectional arrow cannot point to the same dot and needs at least 2 dots

Doubly Linked List

```

#include <iostream>
using namespace std;

struct node
{
    int data;
    node *prev, *next;
} *head = NULL;

```

```
void insertAtHead(int x)
{
    node *tmp = new node();
    tmp->data = x;
    tmp->prev = NULL;
    tmp->next = NULL;

    if (head == NULL) // if the list is empty
    {
        head = tmp;
        return;
    }
    // when the list is not empty
    head->prev = tmp; // establishing link in forward dirn
    tmp->next = head; // establishing link in backward dirn
    head = tmp;      // head should now point at the node we inserted
}

void fwdPrint()
{
    for (node *itr = head; itr != NULL; itr = itr->next)
        cout << "This is " << itr << " having (prev=" << itr->prev << " data=" << itr->data << "
next=" << itr->next << ")" << endl;
}

void revPrint()
{
    node *itr = head;
    if (head == NULL)
        return;          // exit if empty list
}
```

```

    while (itr->next != NULL) // to get to the last node
        itr = itr->next;
    while (itr != NULL)
    {
        cout << "This is " << itr << " having (prev=" << itr->prev << " data=" << itr->data << "
next=" << itr->next << ")" << endl;
        itr = itr->prev;
    }
}

int main()
{
    cout << "[1] Insert At Head\n[2] Print Forward\n[3] Print Reverse\n[4] Quit\n\n";
    int ch;
restart:
    do
    {
        cout << "\nChoice: ";
        cin >> ch;
    } while (ch != 1 && ch != 2 && ch != 3 && ch != 4);
    switch (ch)
    {
    case 1:
        cout << "Enter Element: ";
        int x;
        cin >> x;
        insertAtHead(x);
        break;
    case 2:
        fwdPrint();
        break;

```






```

    case 3:
        revPrint();
        break;
    case 4:
        return 0;
    }
    goto restart;
}

```

Binary Search Trees

- is a binary tree with value of left child node of a parent node always smaller than or equal to the value of parent node and the value of right child node of a parent node always larger than or equal to the value of parent node
- used for effectively searching
- recursion is used for common operations as tree has unidirectional next nodes, we use stack of activation record of function in recursive calls to remember the address of parental nodes
- to check validity, n nodes = $n-1$ connecting links
- depth
 - depth is measured from the topmost node of the tree
 - So root of the tree has depth 0
- height
 - height of a node in a tree is equal to the number of connecting links needed to reach that node from the furthest leaf(a node without any child) under that node
 - height of all leaves is 0
 - height of a tree is height of the topmost node of the tree
 - height if an empty tree is -1

- height of a tree with one node is 0 and so on
- height of a tree with n nodes is n-1 which is equal to the number of connecting links
- max number of nodes for a node at height h $= (2^h) - 1$
- a binary tree for which any node either has 2 or 0 nodes is called a perfect binary tree and height of a perfect binary tree $= \text{floor}(\log_{\text{base2 of } n})$ where n is number of nodes
- cost is proportional to the height of the tree
 - the height is lesser if the tree is denser and a tree is denser if it is a perfect binary tree
 - more a binary tree is perfect, denser it is
 - linked list is the most imperfect binary tree as every node will have one node
 - tree operations are $O(\text{height})$ so for a perfect binary tree $O(\log_{\text{base2 of } n})$ and $O(n)$ for a linked list
 - for tree $O(n)$ is worse than $O(\text{height})$ as $\text{height} = \log_{\text{base2 of } n}$
- try keeping the tree balanced, a balanced tree is one for which $(\text{ht of left child sub tree of parent}) - (\text{ht of right child sub tree of parent}) = 0 \text{ or } 1$
-  binary tree array
-  why use binary search tree
- in a binary search tree at every step you compare the value you are looking for with the value at the node and if it is match you stop
 - else you discard left or right sub tree after comparing the value you are looking for with the value at the current node
-  this is insertion

```
#include <iostream>
#include <queue>
using namespace std;

struct node
```

```

{
    node *left;
    int data;
    node *right;
} *rootptr = NULL; // this points to the root node and is not the root node itself

node *getsetNode(int data)
{
    node *newNode = new node();
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

node *insert(node *rootptr, int data)
{
    if (rootptr == NULL) // breaking condition for recursion
        rootptr = getsetNode(data);
    // these ifs below help to locate the position for new node in the tree and the new node can only
    be inserted if the left or right of another node points to NULL
    else if (data ≤ rootptr->data)
        rootptr->left = insert(rootptr->left, data);
    else
        rootptr->right = insert(rootptr->right, data);
    return rootptr;
}

/**
 * for deleting a node which has 2 child subtree we find the min of right subtree and replace it with
 the node to be deleted and remove the duplicate in right tree
 * max of left is also allowed
 */

```

```

bool search(node *rootptr, int target)
{
    if (rootptr == NULL)
        return false;
    else if (rootptr->data == target)
        return true;
    else if (target <= rootptr->data)
        return search(rootptr->left, target);
    else
        return search(rootptr->right, target);
}

/**
 * to find the ht of the tree: O(n) where n is the number of nodes
 * pseudo code
 * findht(root){
 *     if root is null return -1 // if we find a leaf node its ht will be 0 which is -1 + 1
 *     leftht=findht(root->left)
 *     rightht=findht(root->right)
 *     return (max of leftht and rightht)+1
 * }
 */

int findht(node *rootptr)
{
    if (rootptr == NULL)
        return -1;
    int leftht = findht(rootptr->left);
    int rightht = findht(rootptr->right);
    return leftht > rightht ? leftht + 1 : rightht + 1;
}

```

```
}
```

```
/**  
 * bfs uses queue. s soon as we find a node we add its address to the queue  
 * then u access it and pop the last node from queue and enqueue its child nodes  
 * traversal is completed when the queue is empty  
 * time complexity:  $O(n)$   
 * space complexity:  $O(n)$  in perfect tree or  $O(1)$  in linked list  
 */
```

```
void printBFS(node *root)  
{  
    if (root == NULL)  
        return;  
    queue<node *> q;  
    q.push(root);  
    while (!q.empty())  
    {  
        node *current = q.front();  
        cout << current->data << " ";  
        if (current->left != NULL)  
            q.push(current->left);  
        if (current->right != NULL)  
            q.push(current->right);  
        q.pop();  
    }  
    cout << endl;  
}
```

```
int main()  
{  
    cout << "[1] to insert\n[2] to search\n[3] to show ht\n[4] BFS print\n[5] to quit\n\n";
```

```
int ch;
while (true)
{
    do
    {
        cout << "Enter choice: ";
        cin >> ch;
    } while (ch != 1 && ch != 2 && ch != 3 && ch != 4 && ch != 5);
    switch (ch)
    {
        case 1:
            cout << "Insert: ";
            int ins;
            cin >> ins;
            rootptr = insert(rootptr, ins);
            break;
        case 2:
            cout << "Enter Query: ";
            int query;
            cin >> query;
            if (search(rootptr, query))
                cout << "Found" << endl;
            else
                cout << "NOT Found" << endl;
            break;
        case 3:
            cout << "Ht of BINARY SEARCH TREE IS " << findht(rootptr) << endl;
            break;
        case 4:
            printBFS(rootptr);
            break;
    }
}
```

```


        case 5:
            return 0;
        }
    }
}


/**
 * binary tree traversal is visiting all the elements of the tree exactly once
 * 2 ways
 * bfs is visiting all nodes at a level(top to bottom) from left to right
 * dfs if we are at a child then we completely visit the entire subtree of that child
 * 3 ways
 * preorder    root→left subtree→right subtree short form data left right dlr
 * inorder     left subtree→root→right subtree short form left data right dlr
 * postorder   left subtree→right subtree→root short form left right data dlr
 * data means read or access the data before moving to the next subtree
 */


```

Graphs

- linked list and trees are also graphs
- graphs have no restrictions on connecting links between nodes
- a graph is represented as $G=(V,E)$ where G is an ordered pair of vertices and edges
 - vertices are nodes
 - an edge is the connecting link between two nodes
 - an edge can be directed(one way) or undirected(two way)
 - an edge can have weights like distance between two cities
 - when graph is weighted we cannot just rely on the number of edges between 2 vertices to find the shortest path as we need to consider weights
 - all weights for an unweighted graphs are 1

- graph can have self loops meaning vertex has an edge pointing to itself
 - like how many websites have links to the same page and that is an example of self loop
 - refresh page on the browser leads to the same page
- two vertices can have multiple edges between them example multiple airlines fly between 2 same airports
 - a simple graph has no self loop or multiple edges
- for a simple directed graph the max number of edges for n vertices $= n(n-1)$
- for a simple undirected graph the max number of edges for n vertices $= n(n-1)/2$
- a denser graph has number of edges closer to the max limit else its sparse
 - for a dense graph we use adjacency matrix and for sparse we use adjacency list
- path
 - simple path when vertices/edges are not repeated
 - if repeated it is a walk
 - closed walk when start and end is the same vertex also called cycle
 - trees are acyclic
 - singly and doubly linked lists are acyclic but we also have cyclic/circular linked list where the last node points to the 1st node (not the head)
 - trail when edges are not repeated but vertices are
- representing graph in memory
 - using 2 vectors
 - one stores the content of the vertex called vertex list
 - and edge list which is a vector of a struct and stores 2 integers that indicate a link between the same indices corresponding to the vertex list (and weight also if needed)
 - we know every vertex stored in the vertex list vector has an index
 - we can imagine that an edge connects these indices instead of pointing to them
 -  creating a graph

- Common Operations
 - finding adjacent nodes, checking if 2 nodes are connected is $O(\text{number of Edges or } |E|)$ which is nearly $O(n^2)$ as we know the formula for max number of edges from the number of nodes
 - $O(n)$ is acceptable which is $O(\text{number of vertices or } |V|)$ for graphs Using Adjacency Matrix and Adjacency List
 - Adjacency Matrix of order $|V| \times |V|$
 - better for dense graphs because we will store many 0s for a sparse graph
 - huge storage footprint
 - we have a vertex list of length equal to the total number of vertices in the graph
 - if the length of vertex list be p then the adjacency matrix is $p \times p$
 - for any element at index i of the vertex list, row i of adjacency matrix has values 0 or 1 for every column
 - for column number c and row number r of the adjacency matrix if the value is 1 it means that the element at index r of the vertex list is connected to the element at index c of the vertex list in the graph and 0 means it is not connected
 - for any undirected graph the adjacency matrix will be symmetric so $A_{ij} = A_{ji}$ so we can omit upper or lower triangular matrix with diagonal of matrix as the hypotenuse
 -  adjacency matrix of a graph
 - to find adjacent nodes we do a linear search on the vertex list and then do a linear search on the row of adjacency matrix corresponding to the index found in vertex list
 - which is $O(V+V) = O(2V) = O(V)$
 - to find if 2 nodes are connected or not we can pass their indices i and j and we just need to find the value of $A[i][j]$ in the Adjacency matrix and see if it is 0 or 1 which is $O(1)$
 - if the alphabets are given then we need to do a linear search to find the index in the vertex list making it $O(V)$
 - this can be avoided by using hash map so it will always be $O(1)$

- for weighted graph replace 1 with edge weight and 0 with INT_MAX
- for facebook, using adjacent matrix optimally means for 1 billion users everyone is a friend of everyone else which is impossible so lot of memory is wasted in storing who is not a friend of whom which is redundant as we can figure that out if we just knew who is a friend of who which is infact more important
- So we use Adjacency List which is an array of pointers
 - less memory as space complexity is $O(\text{number of edges})$ as $|E| \ll |V|^2$
 -  Adjacency List
 - we need to perform linear search or binary search on the adjacency list so for finding if 2 nodes are connected $O(|V|)$ or $O(\log |V|)$ which was $O(1)$ in Adjacency Matrix
 - finding adjacent nodes(all the neighbors of a node) is $O(v)$ same as Adjacency Matrix
 - Adjacency Matrix used when the graphs is dense and $|V|$ and $|E|$ are small otherwise Adjacency List is practically preferred over Adjacency Matrix most of the time
- adding a new edge
 - for adjacency matrix flip a zero to one
 - for a list if we are using array of pointers we need to create a new array and copy previous elements into it and then add the new one
 - so use linked list instead? yes and each pointer in the array points to the head of a linked list and the node will store weight if the graph is weighted
 - Adjacency List always uses Linked List and not array or vector