Templates

Generic Programming

It is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.

In C++, templates are used to achieve generic programming.

A template can be used to create a family of classes and/or functions.
A template can be considered as a kind of macro. When an object of specific type is defined for actual use, template definition for that class is substituted with the required data type. Since template is defined with a parameter they are also called **parameterized classes or functions**.

They are foundation of generic programming which involves writing code in a way that is independent of any particular type.

(2) Function Template

SYNTAX

```
template < class T >
returntype functionname ( arguments of type T )
{ ... }
```

Note template <typename T> can be used interchangeably with template <class T>. Both are equivalent

Note For multiple parameters template < class T1, class T2, ----> can be used

Note Template function may be overloaded either by template functions or ordinary functions of its name.

Steps for exact unique match:

1) Search ordinary fun^n for exact match
2) Search template fun^n that could be created with exact match
3) Type Conversions / Type promotions on Ordinary functions

⇒ Automatic conversions on template functions are not applied for exact match.

Note If member functions of a class are to be made generic (template func") then template need to be defined before their function body. { If declared inside class & outside class both cases }

eg

```
template <class T>
vector<T> :: add( T a)
    { ---}
```

{ Here membership label is also classname < typename> :: }

Note : Non-type Template Arguments

We can use arguments such as int, string, function names & built in types also in templates.

eg

```
template < class T, int size>
class array
{
    T a[size];
    ---
};
array <int, 10> a1;
// integer array of 10 size
```

(2) Class Template

SYNTAX

```
template <class T>
class className
{---};
className <type> objName;
```

Process of creating a specific class from a class template is called instantiation.

Note Compiler will perform error analysis only after instantiation takes place. Thus debugging an ordinary class before converting it into template class is advisable.

Note Like in function template, here also, we can pass multiple parameters & non-type arguments to template.

STL

* ADVANTAGES

- easy to use
- saves implementat^n time
- Optimized implementation
- Reusability

TEMPLATES

1) Function Templates

template <typename T>

eg
```
template<typename T>
T add (T a, T b)
{ return a+b; }

cout << add <int> (10,20);
cout << add <float> (1.5, 2.5);
```

2) Class Templates

template <class T> { Before class body & every member fun^n outside class }

eg
```
template<class T>
class stack{
  T arr[10];
  T top;
public:
void push(T x);
T pop(); };

template<class T>
void stack<T>::push(T x)
{ --- }
template<class T>
T stack<T>::pop()
{ --- }
```

Standard Template Library

Collection of generic (template) classes for data structures and generic (template) functions for algorithms that could be used as a standard approach for storing & processing of data is known as STANDARD TEMPLATE LIBRARY {STL}

Benefits of STL

→ easy to use & allow generic programming
→ saves implementation time & effort leading to high quality programs
→ Optimization in algorithms
→ Reusability of well-written & well-tested components of STL

Note: STL components used in Standard C++ library are defined in namespace std;

Components of STL

- Containers
- Algorithms
- Iterators

These 3 components work in conjunction with one other to provide support to

variety of programming solutions

⇒ "Algorithms employ iterators to perform operations stored in containers"

Containers

⇒ Container is an object that actually stores data. It is a way data is organized in memory. Since it is generic (template classes) it can be customized for differed types of data.

Algorithms

Algorithm is a procedure that is used to process the data contained in the containers. Eg, initializing, searching, copying, sorting, merging, etc

Iterator

An iterator is an object (like a pointer) that points to an element in container. They are used to iterate/move through contents of containers. Thus like pointers, they play a key role in manipulation of data stored in containers.

ITERATORS

⇒ used to access data inside containers

TYPES

Forward only and cannot be saved

- INPUT ITERATOR ⇒ read data from container only eg Keyboard
- OUTPUT ITERATOR ⇒ write data into container only eg printer
- FORWARD ITERATOR ⇒ read/write in single direction eg SLL
- BIDIRECTIONAL ITERATOR ⇒ read/write in both directions eg DLL
- RANDOM ACCESS ITERATOR ⇒ read/write in both directions & take jumps eg vectors, array

Can be saved & Read/Write both

#include <iterator>

(1.) begin() ⇒ return beginning posit of container

(2.) end() ⇒ return after end position of contain

(3.) advance(iteratorname, position value)
⇒ increment iterator position by value
eg advance (ptr, 1) ⇒ ptr++;

SYNTAX
eg (display elements)

```
vector<int> :: iterator ptr;

for( ptr = v.begin(); ptr< v.end (); ptr++)
    cout << *ptr << " ";
```

user defined iterator

eg:

```
template <class itr, class T>
                  → linear search
itr search ( itr, start, itr end, T key)
{
    while ( start <= end)
    {
        if ( *start == key)
            return start;
        start++;
    }
    return end;
}
```

```
auto res = search( v.begin(), v.end(), key);
```

Types of Algorithms

1. Sorting algorithms
2. Searching algorithms
3. Non-Modifying algorithms
4. Modifying algorithms
5. Numeric algorithms
6. Min & Max algorithms

Refer to
cplusplus.com/reference/algorithms

ALGORITHMS

(1) find() ⇒ LINEAR SEARCH O(N) (worst case)

↓

returns iterator to the element if found else address next to last element

eg auto itr = find(arr, arr+N, key);

to find INDEX :- index = itr - arr
or index = itr - v.begin()

to find VALUE :- *itr
{ if element not found
itr = v.end() or arr+N
*itr will give garbage value }

(1) sort() ⇒ hybrid of Quicksort, HeapSort and Insertion sort
⇒ known as Introsort

bydefault → randomized quicksort
if quicksort take more than O(N*logN) → Heapsort
if size of array is small → Insertionsort

```
#include <algorithm>
```

(i) for array int A[N]; sort(A, A+N);

(ii) for vector vector<int> V[N];
sort(V.begin(), V.end()) or sort(V.begin(), V.begin+N)

BY DEFAULT it sorts in ascending order for integers and lexicographical order for strings.

To sort in descending order ⇒
sort(A, A+N, greater<int>());

To sort in other order ⇒
sort(A, A+N, compare)
where compare is comparator function

(2) stable_sort() ⇒ mergesort
⇒ stable sorting algorithm
{ key first in unsorted array remains first in sorted when present more than once }

if additional memory is not available $O(N*\log^2 N)$
if additional memory is available $O(N*\log N)$

(3) binary_search() ⇒ array/vector must be sorted
⇒ O(logn)
SYNTAX: binary_search(startadd, endadd, searchkey)
returns true if found else returns false

- lower_bound()
SYNTAX: lower_bound(startadd, endadd, searchkey)
⇒ O(logn)
Single occurence of key ⇒ returns position of key
multiple occurence of key ⇒ returns first position of key
No occurence of key ⇒ returns posit of next higher no than key.

- upper_bound() ⇒ O(logn)
No/ Single occurence ⇒ position of next higher number
multiple occurence ⇒ position of next higher number
first

(4) reverse (startadd, endadd) ⇒ reverses a vector in [start, end)

(5) *max_element(startadd, endadd) ⇒ find max element
*min_element(startadd, endadd) ⇒ find minimum element

(6) accumulate (startadd, endadd, initialvalue of sum)
⇒ returns sum of vector elements

(7) count (startadd, lastadd, key) ⇒ count the number of occurences of key

(8) distance (firstadd, desiredadd) ⇒ returns distance of desired position from first element
⇒ used in finding index

eg distance (V.begin(), V.end())

(9) rotate(start, middle, end);

eg: arr = {1,2,3,4,5}

rotate(arr, arr+2, arr+5);

⇒ arr = { 3,4,5,1,2 }

↳ left rotates the array 2 times

rotate (arr, arr+5-2, arr+5);
(arr+5-2 = size)

⇒ arr = { 4,5,1,2,3 }

↳ right rotates the array 2 times

TIME COMPLEXITY :- Ω(n) (best case)

(10) next_permutation (start, end);

↳ generates lexicographically next greater permutation.

eg: arr = {1,2,3,4,5}
next_permutation (arr, arr+5);
⇒ arr = { 1,2,3,5,4 }

[first, last)

Traversing all permutations
do { --- } while(next_permutation(arr, arr+N))

Comparator

⇒ Passing function as a parameter to another function

eg

```
bool compare (int a, int b)
{ return a>b; }

void bubble_sort( int a[], int n,
                  bool (&compare)(int a, int b))
{
    for(int i = 1 ; i<=n-1 ; i++)
    {
        for(int j=0; j<=(n-i-1) ; j++)
        {
            if (compare( a[j], a[j+1]))
                swap ( a[j], a[j+1]);
        }
    }
}
```

Pair

#include <utility>

- Declaration :-

  pair < first_datatype, second_datatype > p;

  eg pair <int, int> p;
  pair <int, char> p;
  pair< int, vector<int>> p;
  ↳ a pair whose value is vector & key is int.

- Initializatn :-

  # p.first = value;
  p.second = value;

  # p = pair<int, int> (val1, val2);

  # pair<int, int> p2(p1);
  ↳ p2 = p1;

  # p = make_pair(val1, val2);

- vector of pairs

  vector<pair<int, int>> v;

- array of pairs

  pair<int, int> arr[n];

- Operators

>, >=, <, <=

⇒ by default compares only first

==, !=

⇒ compares both first & second

eg: p1(10, 'A') != p2(10, 'B')

CONTAINERS

(1) SEQUENCE CONTAINERS

- vector
- list (DLL)
- deque
- array
- forward_list (SLL)

Stores elements in linear sequence like a line. Every element is related to each other by its position on the line.

(2) CONTAINER ADAPTORS

- queue
- priority_queue
- stack

They provide a different interface for sequential containers.

(3) ASSOCIATIVE CONTAINERS

{ SORTED DS, SEARCHING - O(logn) }

- set
- multiset
- map
- multimap

Associative containers implement sorted Data Structures that can be quicky searched in O(logN)

(4) UNORDERED ASSOCIATIVE CONTAINERS

{ UNSORTED DS, SEARCHING - O(1) }

- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap

They implement unsorted (hashed) Data Structures that can be searched in O(1) amortized and O(N) worst time.

Containers

Container	Description	Headerfile	Iterator
vector	Dynamic Array	<vector>	Random Access
list	Bi-Directional linear (DLL)	<list>	Bidirectional
deque	Doubly ended queue	<deque>	Random access
set	Unique elements	<set>	B'directional
multiset	Duplicate elements	<set>	"
map	Unique key/value pair	<map>	"
multimap	Duplicate key/val pair	<map>	"
Stack	LIFO	<stack>	No iterator
queue	FIFO	<queue>	No iterator
Priority queue	Min Heap & Max heap	<queue>	No iterator

(1) array {C style} class ⇒ static array

#include <array>

SYNTAX: array <Object datatype, arraysize> arrayname;
eg array< int, N> arr;

OPERATORS

i) at() or [] operator ⇒ to access the element
arr.at(k); or arr[k];

ii) front() ⇒ arr.front() ⇒ returns 1st element of array

iii) back() ⇒ returns last element of array

iv) fill() ⇒ initialize each element of array with a particular value.
⇒ arr.fill(num);

v) swap(array2name) ⇒ swaps array1 elements with array2 elements (of same datatype ~~same~~)
⇒ arr1.swap(arr2);

vi) size() ⇒ returns no of elements in array
max_size() ⇒ maxM no of elements array can hold

vii) empty() ⇒ returns true if array is empty else false.

② Strings

`#include <string>` ⇒ character array {dynamic allocation}

(1) ASSIGNMENT (=) or (string value) or assign()

```
string str; str.assign("Hello");
string str = "Helloworld"; or string str("Helloworld");
string str2 = str1; or string str2(str1);
string str = arr; or string str(arr);
```

(2) CONCATENATION (+) or append()

```
string str3 = str1 + str2;   str3 = str1 + "Hello";
str3 = str1 + arr;   {atleast one must be string}
```

(3) concatenation assignment (+=) or append()

```
str3 += str1;
str3.append(str1);
```

(4) Equality

```
if (str2 == str1)
```

(5) Inequality

```
if (str2 != str1)
```

(6) Less than (<), less than equal (<=), greater than (>), greaterthanequal (>=) {lexicographical comparison}

(7) Subscript ([] operator)

```
str[0];
```

(8) insert (key, stringvalue)

eg `str = "Hello"; str.insert(2, "123");` ⇒ `str = "He123llo";`

9) replace (key, length of characters to be replaced, str value)

Str = "Hello"; str.replace(1,3,"12341"); ⇒ Str = "H12340";

10) erase() ⇒ erase all characters from string

11) find(key) ⇒ returns position of first occurence of key string (otherwise -1)

Str = "HelloStudstud"; str.find("stud"); ⇒ 5

12) rfind(key) ⇒ returns position of last occurence of key string (otherwise -1)

13) str1.compare(str2)
returns 0 if both strings are equal
returns 1 if str1 is greater
returns -1 if str2 is greater

14) size() ⇒ returns no of characters in string (or length())

15) INPUT FUNCTIONS

- getline(cin, str) ⇒ stores inputted string completely (with spaces)
- cin >> str ⇒ stores input string upto a word (till before space)
- push_back() ⇒ input a character at end
- pop_back() ⇒ delete last character from string.

(16) capacity() ⇒ returns max^m characters string can store {before doubling}

(17) resize() ⇒ increases/decreases size of string

(18) shrink_to_fit() ⇒ decreases capacity of string equal to its size

---

include <string.h>

(19) substring ⇒ substr(start, length);

⇒ copy string [start, start + length)

if pos = strlength()
substr(pos) will give
" " {empty} string

eg

s1 = "Geeks"
s2 = s1.substr(1, 3)
⇒ s2 = "eek"

if length is not passed it will take all characters from pos/start

(20) Stringtokeniger ⇒ strtok()

SYNTAX :- char* res = strtok(char *s, char *d);

{ d = delimeter }

eg

```
char * ptr = strtok(s, " ");
cout << ptr << endl;
while (ptr != NULL)
{ ptr = strtok(NULL, " ");
  cout << ptr << endl;
}
```

breaks s into tokens/words separated by " "

{ on first call char array or string is passed }

in subsequent calls NULL is passed

(3) VECTORS

`#include<vector>` ⇒ dynamic array or array list

(1) Declaration/assignment

`vector<int> vect {10,20,30}` or `vect = {10,20,30};`

`vector<int> vect;` ⇒ length/size = 0
`vector<int> vect(5);` ⇒ length/size = 5
`vector<int> vect(5,0);` ⇒ length = 5 with all elements initialized 0
`vector<string> vect("Hello", "world");`
↳ vector of string

- `vector<vector<int>> vect { {1,2} {1,2,3} {3,4,8} }`
  - vector of vectors or 2D Array
  - Row1 Row2 Row3

[Each row can have different no. of columns]

`vector<vector<int>> vect[n][m]` or `vect(n, vector<int>(m));`

vector<vector<int>> vect(n, vector<int>(m,0));

↳ initialize nxm matrix with all elements 0

(2) capacity()
size()
resize(n)
shrink_to_fit()
empty()
→ same as string

(3) erase() ⇒ remove element from specified position or range

v.erase(v.begin()) ⇒ removes 1st element

(4) push_back()
pop_back()
swap()
↳ swap one vector with other vector

insert() ⇒ inserts new elements before the element at specified positn

eg vect.insert(v.begin(), 5)

⇒ insert 5 at beginning

(5) Relational operators (>, >=, <, <=, ==, !=)

(6) subscript [] operator or [][] operator

(7) reserve(size); ⇒ initialize vector with given size to avoid repeated doubling.

④ Stack eg stack<int> s; LIFO

#include <stack>

empty() → returns whether stack is empty
size() → returns no of elements in stack
top() → returns reference to top most element
push(num) → adds num element to top of stack
pop() → deletes top most element of stack

⑤ Queue eg queue<int> q; FIFO

#include<queue>

empty() → returns whether queue is empty
size() → returns size of queue
swap() → swap elements of 2 queue (of same type)
front() → returns reference to first element
back() → returns reference to last element
push() → adds element at end of queue
pop() → deletes first element of queue

⑥ forward_list eg forward_list<int>f; singly linked list

#include<forward_list>

(1) assign()

f.assign({1,2,3});

f.assign(5,10); ⇒ assign value 10 '5 times' in the list.

(2) traversal

```
for (int &a : f)
    cout << a << " ";
```

(3) push_front(num) ⇒ inserts element at first position

(4) pop_front() ⇒ delete first element of list

(5) insert_after()

f.insert_after(f.begin(), {1,2,3})

⇒ inserts elements {1,2,3} after first element

(6) remove(num) ⇒ removes particular element from list

(7) front() ⇒ reference the first element

(8) begin(), end(), swap(), empty()
reverse(), sort()

(7) List eg list<int> l; Doubly linked list

#include <list>

front(), back(), push_front(num), push_back(), pop_front(), pop_back(), empty(), insert(), assign(), erase(), reverse(), size(), sort()

(8) deque eg deque<int> d; Doubly ended queue

#include <deque>

Functions are same as vector, with addition of push and pop operations for both front and back. ⇒ O(N) time

push_back(), push_front(), pop_back(), pop_front()

(9) Set ⇒ Associative container

#include <set>

Red Black Tree

⇒ sorted set of unique objects of type key

⇒ Insertion, Removal, Search — O(logN)

{ cannot update already existing/inserted value }

⇒ for userdefined datatype, use comparator

(i) SYNTAX :- set <datatype> setName;

{ Default in ascending order }

for descending order : set <datatype, greater<>> setName;

eg

INPUT	⇒	SET VALUES
{1,2,4,3,1,2}		{1,2,3,4}

(ii) TRAVERSAL :-

```
set <int, greater <int>> :: iterator itr;
for ( itr = setName.begin() ; itr != setName.end(), ++itr)
    cout << *itr;
```

or

```
for(const auto& e : setname)
    cout << e ;
```

(iii) Assignment ⇒ insert(value);

- setName.insert(value);
- set<int> set1(set2.begin(), set2.end());

(iii) Deletion ⇒ erase()

set = {1,2,3,4,5} ⇒ delete 1,2,3

setName.erase(setName.begin(), setName.find(3));
setName.erase(4); ⇒ delete '4' element

(iv) find()

setName.find(value);
⇒ returns iterator to element value if found else returns setName.end();

if element not found/search unsuccessful

↳ setName.find(value) == setName.end()

(v) size(), max_size(), empty(), swap()

used in dictionary type problems.

↳ self balance BST [AVL / Red black tree]

(10) Map ⇒ Associative container
↳ sorted on basis of key value

⇒ Each key value have a unique mapped value
{ Although 2 keys can have same mapped value }
{ No 2 mapped values can have same key value }

MANY-ONE MAPPING

#include <map>

SYNTAX : map<datatypekey, datatypemapped> mapName;

i) insert()

mapName.insert(pair<keytype, maptype>(keyvalue, mappedvalue));

map1.insert(pair<int,int>(9212195907, 907));

ii) traversal

map<int, int>::iterator itr;
for(itr = map1.begin(), itr != map1.end(), itr++)
cout << itr->first << endl << itr->second;

first ↓ key value
second ↓ mapped value

iii) erase(), find() similar to set
size(), max_size(), empty(), swap()

→ used for searching

(11) unordered_set → not sorted; but unique values — HASHTABLE

- all operations → O(1) time
- worst case O(N) time (bad hash function)
- for userdefinedtype of value, comparator need to be defined

SYNTAX: unordered_set<datatype> setName;

#include <unordered_set>

(i) insert()

unordered_set<string> set1;
set1.insert("Hello");

(ii) find() ⇒ SEARCH IN O(1) time

string key = "would";
if(set1.find(key) == set1.end())
cout << key << "not found";
else cout << key << "found";

(iii) traversal

unordered_set<string> :: iterator itr;
for(itr = set1.begin(); itr != set1.end(); itr++)
cout << *itr;

→ used for counting frequencies

→ unsorted

(12) unordered_map — HASHTABLE

↳ stored elements formed by combination of key value and mapped value.

search, insert, delete — O(1) time

(i) assignment

eg unordered_map <string, int> map1;

```
map1["Hello"] = 10 ;
map1["world"] = 20 ;
```

or `map1.insert(make_pair("world", 20));`

(ii) traversal

```
for( auto x : map1)
cout << x.first << " " << x.second << endl;
```

(iii) find() ⇒ same as unordered_set

Custom Hash function

for key ⇒ custom (user defined class)

eg: class Student { public:
string firstname, lastname;
int rollno;
Student (string f, string l, int no)
{ firstname = f; lastname = l; rollno = no; }

to define collisions in hash function

bool operator == (const Student &s) const
{
return (rollno == s.rollno);
}
};

making funct' const

class HashFn {
public:

size_t operator () (const Student &s) const
{
return (s.firstname.length()
+ s.lastname.length());
}
};

stores size of some memory in bytes & in long unsigned int

```
unordered_map<Student, int, Hashfn> marks;

Student s1("Archit", "Aggarwal", 68);
Student s2("Archit", "Gupta", 69);
Student s3("Akshat", "Jain", 43);

marks[s1] = 24;
marks[s2] = 20;
marks[s3] = 25;

// Iteration over hashmap
for(auto s: marks) {

cout << s.first.firstname << " "
     << s.first.lastname << endl;
cout << "marks = " << s.second << endl;
```

Output :-

```
Archit Aggarwal
24
Archit Gupta
20
Akshat Jain
25
```

Dictionary Problem

⇒ find if a word/string is present in dictionary (vector of strings) or not.

N = no of words/strings M = max^m string length

(1) Brute force ⇒ O(N*M)

· compare string with every string in the vector of strings/dictionary

(2) Hashmap ⇒ O(M)

Insert all words/strings in a single map and then check its presence.

It is slower { in worst case O(N2) where N = no of words/strings } for very large N as many collisions might take place.

(3) Trie ⇒ O(M)

ADV:- Faster than hashmap when N is very large because there are no collision handling in trie.

DISADVANTAGE:- Even when N is small, trie takes a lot of memory space due to multiple hashmaps (for each node) thus space complexity is very high. For small N, simple single hashmap implementation will be better.

TRIE ⇒ Prefix Tree

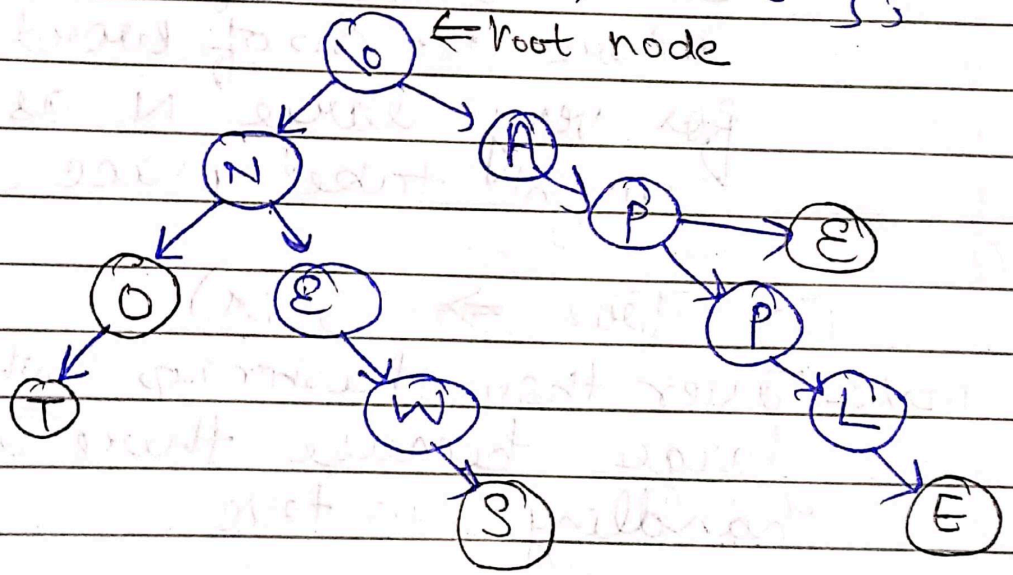
Dictionary Based Data Structure

⇒ Searching if a number / word / string is present in a vector of numbers (phonebook) / vector of strings (dictionary) ⇒ O(M) Time

{String length}

eg

```
Vector<string> arr = { "no", "not", "apple", "ape"
                     , "news" };
```



○ ⇒ Terminal Node

{ root to terminal node forms a word }

○ ⇒ Non-Terminal Node

{ word starting from root ending at this node does not become a valid word }

// Implementation

```
class Node {
public:
unordered_map<char, Node*> children;
bool terminal;
char data;
Node(char d)  // constructor to create new node
{ data = d;
  terminal = false;
}
};

class Trie {
Node* root;
int count;

Trie() {   // constructor to create root node
  root = new Node('\0');
  count = 0;
}

void insert(char *w) {
Node* temp = root;
for(int i=0; w[i] != '\0'; i++) {
        char ch = w[i];
        if (temp->children.count(ch))
            temp = temp->children[ch];
// if node is present then traverse
```

```
else {   // node is not present => create node

    Node* newNode = new Node(ch);
    temp->children[ch] = newNode;
    temp = newNode;
  }
}
temp->terminal = true;
// make last char of word as terminal
   node outside for loop

bool find(char *w)
{
  Node* temp = root;

  for(int i=0; w[i] != '\0'; i++)
  {
    char ch = w[i];
    if(temp->children[ch] == 0)
      return false;
    // if current letter/char
     is not present return not found
    else
      temp = temp->children[ch];
      // traverse the current letter
  }

  // if you reach end of word, check
     if it is a terminal word, if yes
      return found else not found
  return temp->terminal;
}
};
```

(13) Priority Queue — MAX HEAP

→ used to extract elements based on highest priority

→ can be used to implement 'heap sort'

Insert N elements ⇒ O(N) [heapify]

Extract k^{th} largest element ⇒ O(klogN)

Pop one element / Extract largest priority element ⇒ O(logN)

(i) Initialization

```
#include<queue>

priority_queue <int, vector<int>> pqname;
                    OR                  ↳ stores heap as vector
Simply  priority_queue <int> pqname;

                    OR

priority_queue <int, deque<int>> pqname;
                                ↳ stores heap as deque

for min heap :
priority_queue <int, vector<int>, greater<>> pq;
```

(ii) Insertion

```
pq.push(element);
```

(iii) Traversal

(works as heap sort ⇒ returns elements in descending order)

```
while( !pq.empty())
{
    cout << pq.top();
    pq.pop();
}
```

pq.top() → returns root of heap → A[0] → greatest priority element

pq.pop() → deletes largest priority element & modifies max heap

(iv) pq.size() ⇒ returns no of elements in priority queue

Functors ⇒ Functional objects

⇒ They are objects of a class which look/behave like a function

eg:
```
Class A {
public:
void operator()(string str)
  { cout << str << endl; }
};
```
→ operator overloading

A obj; ⇒ object of class A
obj("Hello"); ⇒ looks like function call but actually it is a object with () operator overloaded

{ It is not a function but a class here }

Comparator for Priority Queue of a user defined datatype

eg:
```
class Person { public:
string name;
int age;
Person() { }
Person(string n, int a)
  { name = n; age = a; }
};
```

```
class Compare
{ public:
bool operator()
(Person A, Person B)
{ return A.age < B.age; }
};
priority_queue
<Person, vector<Person>,
Compare> q;
```

Policy Based Data Structure

```
// Header files
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>,
  rb_tree_tag, tree_order_statistics_node_update>
pbds;
```

→ Stored as in set ⭐ { unique and ordered }

True structure

{ other tree structures :
  splay_tree_tag ⇒ splay tree
  ov_tree_tag ⇒ ordered-vector tree }

Functions

⭐ O(logN) time complexity

(1) find_by_order(k)

returns iterator to (k+1)th largest element { 0-based indexing }

(2) order_of_key(k)

if element is present
⇒ returns index of k
ie. no of elements strictly less than k.

if element is not present
⇒ returns index if k would have been present in set
ie no of elements strictly less than k.

eg

```
//header files included

pbds st;
st.insert(10);
st.insert(3);
st.insert(10);
st.insert(4);
```

* st.find_by_order(2) ⇒ 10
{ 3rd largest element }

order_of_key(12) ⇒ 3
{ 3 elements in set less than 12 }