# Array Data-Structure

1. Homogeneous
   a. {10, 20, 30, 40}
   b. {“abc”, “def”, “ghi”}
2. Fixed Size
3. Random Access
4. Continuous

# Declaration Techniques

| Static Allocation | Dynamic Allocation |
| --- | --- |
| Size allocated at compile time | Size dynamically allocated at runtime |
| Allocation is done on stack | Allocation done in heap memory |
| Stack allocation takes less time | Heap allocation takes more time |

# Static Allocation

```
int a[100]
```

# Dynamic Allocation

1. C

   int *a = new int[size]

   free(a)
2. C++

   int *a = (int *) malloc(size * sizeof(int))

   delete []a

# Vector

1. Variable Sized Arrays
2. Allocated on heap

# Operations on Vector

```
vector<int> v(3, 10);
                or                          // {10, 10, 10}
vector<int> v = {10, 10, 10}
v.push_back(20);                            // {10, 10, 10, 20}
int x = v.back();                           // 20
int y = v[2];                               // 10
v.pop_back();                               // {10, 10, 10}
```

# Question - I

1. Figure out which statement takes least time
    a. int a[1e6];
    b. size = 1e6; int *a = new int[size];
    c. vector<int> a(1e6);

# Map and Set

1. Vector has insertion time of O(1) and find time of O(n).
2. Map and Set have both insertion and find time of O(log(n))
3. Internally C++ maps/sets are implemented using Red-Black Tree

# Operations on Map and Set

| Map | Set |
| --- | --- |
| map<int, int> mp | set<int> s |
| mp.insert({10, 3}) or mp[10] = 3 | s.insert(10) |
| map<int, int> :: iterator it = mp.find(10) | set<int> :: iterator it = s.find(10) |
| map<int, int> :: iterator it = mp.lower_bound(10) | set<int> :: iterator it = s.upper_bound(10) |
| mp.erase(10) or mp.erase(it) | s.erase(10) or s.erase(it) |

*Iterators are a means to loop over non-contiguous memory locations of stl containers

# Question - II

1. Which of following statement is true

```
set<int> s = {10, 20, 30, 40};
map<int, int> mp = {{10, 1}, {20, 2}, {30, 1}};
```

a. s.find(30) == s.end();
b. mp[10]--; mp.find(10) == mp.end();
c. auto it = mp.lower_bound(30); it != mp.end();
d. auto it = mp.upper_bound(30); it != mp.end();

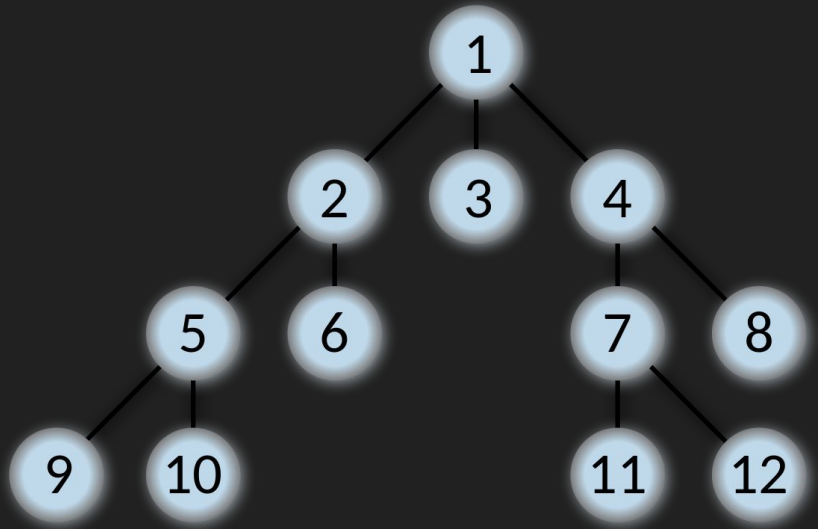
# Queue and Operations

1. First In, First Out.
2. Elements enter queue from back, and come out from front.

a. queue<int> q;
b. q.push(10);
c. int front = q.front();
d. Int back = q.back();
e. q.pop();

# Queue in BFS: Just a Glance

```
queue<int> q;
q.push(root);
while(!q.empty()){
        int node = q.front();
        q.pop();
        for(auto &child: g[node]){
                q.push(child);
        }
}
```



# Priority Queue and Operations

1. Element with highest priority is popped first.
2. Implemented internally using heap.

a. priority_queue<int> pq; // priority: descending order
b. priority_queue<int, vector<int>, greater<int>> pq;
   // priority: ascending order
c. pq.push(10);
d. int x = pq.top();
e. pq.pop();

# Stack and Operations

1. Last In, First Out
2. Only Top is accessible

a. stack<int> st;
b. st.push(10);
c. int top = st.top();
d. st.pop();

# Check Valid Parentheses

1. Given a sequence of opening and closing parentheses, check whether it is valid (balanced) or not

(((()())())) ()(() ((()))())

# Question - III

1. What will be the final stack contents for bracket sequence