Kalapi
Page No.
Date

Lecture 2

Container: set : Stores only unique elements & in sorted manner

Declare : set<int> st;

Function: st.insert(2); // O(logn) time

X No random access st[0] X

access by iterators = ~~st.begin()~~ *st.begin()

Function:- st.erase(st.begin()) // O(logn)

↳ iterator to delete

// O(logn) st.erase(st.begin(), st.begin+2); [)

deletes first 2 elements.

~~Attempts for deleted~~

st.erase(5); // O(logn)

Declare:- set<int> st = {1,5,7,8};

- set<int> st1(st.begin(), st.end());

Function:- auto it = st.find(7); // O(logn)

If 7 exists in st, it points to 7

If 7 doesn't exist in st, it = st.end();

Function: st.emplace(6); // emplace faster than insert.

Function: st.size(); // gives size of set.

Function: st.erase(st.begin(), st.end()) // delete all elements.

Container: unordered_set<int> st;

1) No guaranteed ordering of elements.
2) Average time complexity is O(1)
3) Worst case time complexity is O(setsize) & (O(n))

Worst case depends on hashes generated.

Kalapi
Page No.
Date

→ Thumb Rule:-
If no need of storing elements in ascending order go for unordered_set(), if you get TLE → switch to set()

Container: multiset<int> ms; stores duplicates also. stores all elements in sorted manner.

function: ms.erase(2) // erase all instances of 2 in multiset

function: auto it = ms.find(2); // returns iterator to the first instance of 2.

O(logn) operations

function: ms.erase(ms.find(2)). ← Only delete 1 instance
ms.erase(ms.find(2), ms.find(2)+2) ← delete 2 instances of 2.

function:- ms.count(2); // No. of instances of 2

Container:- Map:- Key value pairs.
Only unique keys.; O(logn) operations

Declare: map<string, int> mpp;
mpp["John"] = 100;
function: mpp.emplace("Tiger", 40);

function:- mpp.erase("John");
" mpp.erase(mpp.begin());
" mpp.clear();
" mpp.erase(mpp.begin(), mpp.begin()+2);
" auto it = mpp.find("John");
mpp.empty() → true if mpp empty; false otherwise

Kalapi

Page No.

Date

fun(key): mpp.count("raj") // One in case of map if "raj" key exists, zero otherwise.

Pair:- pair<int,int> pr;
pr.first = 1;
pr.second = 2;

it → pointer to the pair
it → first (or) (*it).first

Printing map:

for(auto it : mpp)
{
cout << it.first << " " <<
it.second << endl;
}

Pair

for(auto it = mpp.begin();
it != mpp.end(); it++)
{
cout << it → first << " " <<
it → second << "\n";
}

Container :- unordered_map<int,int> mpp;
// Random order ; O(1) in almost all cases
// O(n) in the worst case.

PAIR; pair<int,int> pr = {1,2};
pr = make_pair(1,2);
pair<pair<int,int>,int> pr = {{1,2},3};
pr.first.first = 1 pr.first.second = 2
pr.second = 3

unordered_set < pair<int,int> > ✗ Not Allowed.
unordered_map < pair<int,int>, int > ✗ Not allowed.
map < pair<int,int>, int > ✓ Allowed
set < pair<int,int> > ✓ Allowed.
Vector < pair<int,int> > ✓ Allowed

Container :- multimap<string,int> mpp;
stores duplicate keys also.
Sorted fashion.

Kalapi

Page No.

Date

No iterator concept in Stack - Queue

Data structures — Stack and Queue

stack <int> st; Last In First Out D.S.

// Pop, top, size, empty, push and emplace // O(1) operation

Delete :- No clear function No clear() function.

To delete entire stack ⇒ while (!st.empty()) // Linear time

{

st.pop();

}

stack <int> st;

cout << st.top(); or st.pop() on empty stack ⇒ throws error.

Usually runtime errors in online judges

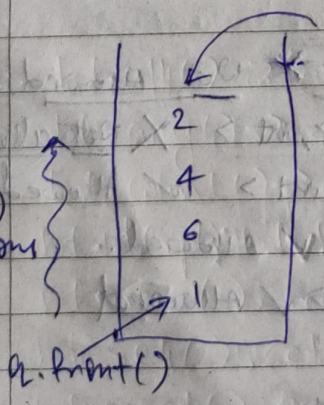
Imp :- Always use top() & pop() with a check condition

```
if( !st.empty())
{
  cout <<st.top(); or st.pop();
}
```

Data structures — Queue First In First Out D.S

queue <int> q;

// push, emplace, front, pop, size, empty // O(1) operations



q.push(1) (6) (4) (2)

clear

To clear whole queue use while loop.

while (!q.empty()) // linear time

{

q.pop();

}

Kalapi
Page No.
Date

No iterator concept

Data Structure

priority-queue (Most important). // Use concept of HeapSort
→ stores all elements in sorted order.
→ O(log n) operations.
// push, size, top, pop, empty

priority-queue <int> pq; → MAXIMUM PRIORITY QUEUE :-

pq.push(6)	6 ← top	
" (1)	4	pq.top() operation
" (2)	2	
" (4)	1	

↳ To store element like minimum priority queue, negate elements while inserting.

pq.push(-6)	-1 ← top	
" (-1)	-2	// while fetching do negation again :-
" (-2)	-4	
" (-4)	-6	cout << (-1) * pq.top();

Data structure:

* Minimum priority Queue :-

priority_queue <int, vector<int>, greater <int>> pq;
(greater means Ulta)

pq.push(6)	1 ← top()	
pq.push(1)	2	top operations
pq.push(2)	4	
pq.push(4)	6	

* for min. priority queue of pairs, replace int with pair.
priority_queue <pair<int,int>, vector<pair<int,int>>, greater <pair<int,int>>>

Kalapi
Page No.
Date

#. Containers

Deque : Doubly ended Queue

```
deque<int> dq;
// push_front  // pop_back
// push_back   // begin, end, rbegin, rend
// pop_front   // size   // empty
               // clear  // at
```

All functions of vector

Container List

```
list<int> ls;
```

All operations of deque

// remove() → O(1) operation.

eg. ls.push_back(1)
" " (2)
" " (0)

ls.remove(2); // O(1) operation.

Containers: Bitset → 1 bit

int :- 32 bits.

char :- 8 bits.

Declare: `bitset<10> bt;` → 10 bits.

Input: `cin >> bt;` // 1011011100

functions :-
- bt.all() → returns true if all bits are set, false otherwise
- bt.any() → " " " atleast one bit set, false "
- bt.count() → returns no. of set bits
- bt.flip(); bt.flip(index); 0→1 or 1→0
- bt.none(); returns true if none bits set, false otherwise
- bt.set(); sets all bits; bt.set(index); bt.set(index, 0or1)
- bt.reset(); make all bits 0; bt.reset(index)
- bt.size(); returns size of bitset
- bt.test(index); true if bit at index is set, false otherwise;