1. A) i) Stack is a data structure that operates on the LIFO method i.e Last In First Out, it also has a fixed size.

Queue is a data structure that operates on the FIFO method i.e First In First Out, it also requires fixed size input.

In Stack to add, remove and view an item you perform the following operations; Push, Pop, Peek. In Queue it is; Enqueue, Dequeue, Front (View first item) and Rear (View last item)

ii) Abstract Data Type is a conceptual idea/model that defines the set of operation we can perform on data structures, without specifying how this operations are implemented or how it is organized in memory.

Example of ADT: a List, Stack and Queue.

List operation; It provides ordered way to store, access, and modify data; get(), insert(), remove()

Stack operation; This allows elements to be added and removed only from one end; push(), pop(), peek()

Queue operation; This allows elements to be inserted at one end (rear) and removed from the other end (front); enqueue(), dequeue(), front()

B)

#[derive(Debug)]

struct Node {

value: i32,

next: Option<Box<Node>>,

}

#[derive(Debug)]

struct LinkedList {

head: Option<Box<Node>>,

}

impl LinkedList {

fn new() -> Self {

LinkedList { head: None }

}

fn insert\_at\_end(&mut self, value: i32) {

let new\_node = Box::new(Node {

value,

next: self.head.take(),

});

self.head = Some(new\_node);

}

fn print(&self) {

let mut current = self.head.as\_ref();

while let Some(node) = current {

print!("{} -> ", node.value);

current = node.next.as\_ref();

}

println!("None");

}

}

C) Advantages of Singly Linked List over Dynamic Arrays:

Insertion/Deletion – In a singly linked list the insertion and deletion operations are much more efficient. This is because each node contains a pointer to the next node and the overall complexity is (O(1)). This is as opposed to dynamic arrays where it is a contiguous block of memory that resizes dynamically. Insertion and Deletion is much more costly with the (O(n)) complexity

Resizing – In a singly linked list there is no need of resizing, this is because nodes are added dynamically according to the user’s need. This is in contrast to dynamic arrays where resizing involves allocating a large block of memory and copying elements. This can be resource intensive.

1. A) (i)

(ii) Inorder Traversal operates on Left -> Root -> Right

* Start at 10,
* Then visit 20
* Then go to 30
* Then jump to Root i.e 40
* From the root jump to 60
* Then go to 70
* And drop down to 80
* Finally end on 25

(B) hash table of size 7 (indices 0 to 6) and a hash function h(key) = key % 7.

Insert 44, 45, 79, 55, 91, 18.

1st = 44 % 7 = 2 so it will go in slot 2

2nd = 45 % 7 = 2 it can’t go in slot 2 so will check the previous slot 1, which is 1 and put it there

3rd = 79 % 7 = 3 place it in slot 3 cause its empty

4th = 55 % 7 = 6 goes straight into slot 6

5th = 91 % 7 = 0 goes directly into slot 0, no collision

6th = 18 % 7 = 4 goes into slot 4, no collision

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 91 | 45 | 44 | 79 | 18 |  | 55 |