Data Structures using C

Madhuri B

KERNEL MASTERS

Introduction

- **Data Structure in C**. **Data structures** are used to store **data** in a computer in an organized form.
- In **C** language Different types of **data structures** are; Array, Stack, Queue, Linked List, Tree

Arrays

- Array is a linear data structure.
- Array is collection of similar data type.
- Array elements are stored in consecutive memory locations.
- We can access any element of an array using indexes.
- Deleting or Inserting an element at the middle of an array requires lot of programming.

Stack

- Stack is a linear data structure.
- You can access only the top most element that is added at the last.
- This is called Last In First Out list.
- Insertions and Deletions are allowed only at one end, called top.
- The insertion operation is called push.
- Deletion operation is called pop.
- Can be implemented by using arrays and linked lists.

Creation of stack using array

- Take a one dimensional array stack_arr[].
- Take a variable top, which keeps track of the index of the top most element in the array.
- Initialize top to -1, to indicate the stack is empty.
- For push operation , value of top is increased by 1 and then the new element is added at that index. The new top value should be returned.
- For pop operation, first the element at the top index is popped, and then top is decreased by 1. The new top value should be returned.
- The functions push and pop should check for overflow and underflow.

Queue

- Queue is a linear data structure.
- The first item you have added is the first item that can be retrieved.
- Called First In First Out list.
- Insertions happen at the rear end and deleted at the other end, called front end.
- Insertion operation is called enqueue.
- Deletion operation is called dequeue.
- Can be implemented using arrays and linked lists.
- Types available : circular queue,Deque,Priority Queue,

Creation of Queue using Arrays

- Take two variables, rear and front.
- both rear and front are initialized to -1.
- For insertion in queue, value of rear is incremented by 1, and the element is inserted at the new rear index.
- For deletion, the value of rear is incremented by 1, and the element at front position is deleted.
- When rear = MAX-1, cannot enque.
- When front == rear or front == MAX-1, cannot deque.

Circular Queue

- To overcome the limitation of array implementation ,we can implement it as Circular Queue.
- When value of rear is MAX-1. then instead of incrementing rear, we will make it zero and then perform insertion.
- Similarly when value of front becomes MAX-1, it will be reset to zero.

DEque

- Double Ended Queue.
- Elements can be inserted or deleted at either end of the list.
- Two variables needed ,front, rear.
- 4 operations needed :
  - Insertion at the front end
  - Insertion at the rear end
  - Deletion from the front end
  - Deletion from the rear end

Priority Queue

- Every element of queue has some priority.
- Element with higher priority will be processed before the element with less priority.
- If two elements have same priority then the FIFO rule will follow.
- There are two ways of implementing :
  - Insertion is simple, like in Queue. While deleting need to search for high priority and then delete.
  - Insertion is costly, because need to find where to insert first. Deletion is simple because the front end is to be deleted always.

Linked List

- Linked list is a linear list of nodes.
- Nodes are not stored in contiguous locations.
- You can add a node or delete any node from any point of the list(beginning, middle or end).
- Accessing the elements is not straight forward like in Arrays as it involves traversing through all the items.
- Can grow and shrink as needed.
- No need to define the initial size.
- Uses Dynamic memory allocation.

Linked Lists

- Each element is a dynamically allocated node.
- Each node has two parts – 1)Data 2)Link
- Data part contains the member variables to store the actual data.
- Link part is a pointer that points to the next node in the list.
- For the last node, the Link pointer will be a null.
- A local pointer variable should always be pointing to the head of the list. Otherwise we cannot use the linked list as the elements are not stored side by side.
- We have to use a self referential structure to represent a node as it has to contain data and link both.

Self Referential Structure

- A structure that contains a pointer to a structure of same type.

```
struct node
{
type1 member1;
type2 member2;
struct node *link;
};
```

- link is the pointer member that stores the address of the next node.

Array

- Array is a sequential representation of a list.
- Array elements are stored in consecutive memory locations.
- Data can be accessed using Index.

- Data access time is equal for any element as it is done through index.
- Can be referred to by the array variable name that represents the base address.
- Can be created either using static or dynamic memory allocation.
- Has a fixed size while creation.
- Within the existing size limits, we can add new elements at the end of the array.
- Even if an element is made null, the array size doesn’t change.
- Inserting a new item or deleting an item is tedious, involving element shifting.

Linked List

- Linked List is a linked representation of a list.
- Linked List elements are stored in non-consecutive memory locations.
- Data is accessed by traversing to each node through the links.
- Data access time of a node depends on the distance of the node from head.
- A temporary pointer will be pointing to the address of the first node,called head.
- Can be created using dynamic memory allocation only.
- Size is not fixed while creating.
- There is no size limit. We can keep adding or removing elements
- We can completely remove a node from the list and free the memory allocated for the node.
- We can remove or insert items at anywhere in a linked list easily.

How to create a student Linked List

- Define the Self Referential Structure struct student.
- Declare a head pointer of base type struct student initialize it to NULL.
  struct student * head = NULL;
- Take another traversing pointer current and assign it to NULL.
  - struct student *current = NULL;
- In a for loop ,that loops through the number of nodes to be created :
- create a temporary pointer, node, and create the structure element dynamically using malloc. Typecast it to the struct student type.
  - struct student *node = (struct student *)malloc(sizeof(struct student));
- set the next pointer of node to NULL:
  - Node->next = NULL;
- Allocate the ID,student name and marks to the data elements of node.
- If the node created is first node (if head == NULL) ,then set current and head pointers to node:
  - current = node; head = node;
- IF the node is not first node, then Link current node to newly created node, and then move it to the newly created node.
  - Current->next = node; current = node;
- Go to the next iteration.
- After your loop completes, your list is ready. Head is your handle for the entire linked list.

How to traverse a linked list

- Place the current pointer at head.
- While current pointer is not null,
  - Print the student details from the node located at current pointer.
  - Move the current pointer to the next node and go for the next iteration.
- When current pointer becomes null, which means last node is crossed, stop the loop.

How to add a new student to the list

- Point the current node to the head.
- While current->next is not null, keep moving to the next node. When current->next becomes null, that means, current node at last node now.
- Create the new node to be appended.
- Take the data elements for new node.
- Set node->next to NULL.
- Set current->next to node.

How to delete a student from list

- Take the search key from user (ID of the student here).
- Take current and previous pointers.
- Initialize current to head.
- Traverse through the list until you find the node with the ID by moving current to the next node.
- Point the next pointer of previous node to the next pointer of current node.
- Free the current node.

Creation of stack using linked list

- Beginning of a linked list should be taken as the top.
- For push operation, a node will be inserted at the beginning of the list.
- For pop operation ,first node of the list will be deleted.
- There is no size limit. So you just have to check for the memory availability during malloc before pushing.
- Before popping, check if the top node is not null.

Linked List Implementation of Queue

- We need two pointers.
- Beginning of a linked list is front.
- End of a linked list is rear.
- Enque operation will add a new item at the end.
- Deque operation will delete a item from the front.
- Initially both front and rear are initialized to NULL.
- There is no size limit. So we can keep adding new items and deleting the items.
- The limitation we have in array implementation is overcome with linked list implementation.

Reverse a Single Linked List

- Take three pointers, prev, current & next.
- Initialize all three pointers to head.
- Set current = current -> next
- Set next = current -> next
- Now, prev, current and next are pointing at first three nodes in sequence.
- As long as(next != null)
- Set current -> next = prev
- Move further. Prev = current. current = next. next = next->next.
- Repeat the loop.
- Once the loop is over, point the head->next to null.
- Make head = next (move the head node to last node, as it is reversed).

KERNEL MASTERS

How to find middle element of a linked list in one pass

- Maintain two traversing pointers, current and middle.
- Initialize both to head.
- After passing every two nodes using current, increment middle once.
- When current is at the last node, middle will be pointing to the middle node.
- This solution works if there are odd number of nodes.
- Incase there are two elements at middle (even number of nodes), you end up with first element in second half when current becomes null.

How to find loop in linked list

- Two runners running in a circular jogging track will meet at some point though they start at different points and are running at different speeds.
- Similar to finding middle element.
- Take two pointers, start at the head. One is slow pointer, another is fast pointer.
- Slow pointer will be moved next once, after the fast pointer is moved next twice.
- At one point, both pointers will become same, if there is a loop.

How to remove the loop in a linked list

- After finding the loop,we have to find where the loop is starting, so that we can remove the loop.
- Once slow and fast pointers meet at one point, keep one pointer at that meeting point and make the other pointer point to the head.
- Start moving both pointers one node at a time.
- The point where both nodes meet will be the beginning of the loop.
- You have to keep a previous pointer that keeps track of the pointer previous to the one inside loop.
- Once both pointers meet, make the previous pointer->next to null.

Find nth element from last in one pass

- Take two pointers.
- Start both at head.
- Move one pointer to the n+1th element from head.
- Now one pointer is at first element. Second pointer is at nth element. Both are at a distance of n.
- Now move both by one node each until the second pointer reaches NULL.
- Now the first pointer is the nth element from last.

Find intersection of two linked lists

- Traverse one linked list from head to end node.
- Link end node to head and make it a circle.
- Now both linked lists together form a single linked list with loop.
- Now find the start of the loop.

Given only a pointer/reference to a node to be deleted in a singly linked list, how do you delete it?

- copy the data from the next node to the node to be deleted and delete the next node.
- solution only works if there are no outside references to the list.
- solution works with the last node only if the data structure is augmented with a specific "last node" element.

Circular Linked List

- The last node points to the Head node, instead of NULL.
- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- So instead of checking for NULL, we can check for Head as a ending point.
- Useful for some of the operating system features like scheduling algorithm implementations.

Double Linked List

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **Prev** – Each link of a linked list contains a link to the previous link called Prev.

Search Algorithms

Searching is the process of finding the location of the specified element in a list.

- Sequential Search
- Binary Search

Linear / Sequential Search

- Start at the beginning and walk to the end testing for a match at each item.
- Simplicity
- Cannot go wrong
- For(i from 0 to n-1)
  - if i th element matches the search key
    - Return success
  - Else
    - Return false

Binary Search

- This is used when the list is sorted.
- Starting at the middle, we need to find in which half the item may be found.
- Highly efficient.
- The search needs to be performed untill the both ends flip.

Binary Search procedure

- Given an array A of n elements with values A[0], A[1], ..., A[n−1], sorted such that A[0] ≤ A[1] ≤ ... ≤ [An−1].
- If we have to search for T, the following function uses binary search to find the index of T in A.
- Set L to 0 and R to n − 1.
- If L > R, the search terminates as unsuccessful.
- Set m (the position of the middle element) to the floor (the largest previous integer) of (L + R) / 2.
- If Am < T, set L to m + 1 and go to step 2.
- If Am > T, set R to m − 1 and go to step 2.
- Now Am = T, the search is done; return m.

Bubble sort

- Performed by repeatedly swapping the adjacent elements if they are in wrong order.
- For every iteration of the outer loop ,the next biggest element is bubbled to the right most corner.
- For every nth iteration the last n elements are already sorted, so the inner loop always runs n times less everytime.
- If any iteration results in no swappings, the algorithm will stop.

Bubble Sort procedure

- Run a nested loop. The outer loop will loop through n-1 times, if there are n elements to be sorted.
- The inner loop just compares adjacent elements and will swap them if they are not in correct order.
- Starting from the first index, compare the first and the second elements. If the first element is greater than the second element, they are to be swapped.

  Now, compare the second and the third elements. Swap them if they are not in order.

  The above process goes on until the last element.
- The same process goes on for the remaining iterations. After each iteration, the largest element among the unsorted elements is placed at the end.

  In each iteration, the comparison takes place up to the last unsorted element.

  The array is sorted when all the unsorted elements are placed at their correct positions.

Selection Sort

- The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting
- In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.g it at the beginning.

Selection Sort

- The algorithm maintains two subarrays in a given array.
- 1) The subarray which is already sorted.
  2) Remaining subarray which is unsorted.
- The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.

Algorithm

- To sort an array of n elements, take two indexes i & j.
- Start a for loop that runs n number of times with loop variable i.
- Everytime we select a minimum number from the unsorted part, it will be placed at index i. So assign it to min_index.
- Start another loop to run from i+1 till end of loop,with loop variable j.
- Find the smallest element index in this loop.
- Swap the smallest element index element with min_index element.
- Continue with next selection.

Insertion Sort

- Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.
- Loop from i = 1 to n-1.
  ……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

Merge Sort

Merge Sort is a Divide and Conquer algorithm

```
MergeSort(arr[], l,  r)
If r > l
1. Find the middle point to divide the array into two halves:
        middle m = (l+r)/2
2. Call mergeSort for first half:
        Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
        Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
        Call merge(arr, l, m, r)
```

Trees

- Tree is a non-linear data structure that enables easy search, in a limited number of visits.
- Each element is called a **Node**. Every tree has a node with no parents, called root node. Nodes without any children are called Leaf nodes.
- **Level** of any node is defined as the distance of the node from root.
- **Height** of the tree is the total number of levels in the tree / highest level + 1.
- The number of subtrees or children of a node is called its **degree**.

Binary Tree

- Trees in which every node can have 0 ,1 or 2 children, is called a Binary Tree. Each child is either left child or right child.
- Full Binary Tree is a binary tree, where every node has exactly 2 children, and all levels have maximum number of nodes.
- Complete Binary tree is where only the last level contains less number of elements than the maximum but they are inclined only towards left.

Array representation of binary tree

- Every node starting from root is given a serial number, from left to right in a level. And they are stored in an array.
- The nodes which are absent are also given numbers, and the corresponding indexes in array are left empty.
- Array representation may waste memory.

Linked List representation of Binary Tree

- Every node in a Binary tree requires two links, to both its left and right childs.
- So need to declare a self referential structure with 2 members referring to the same data type as links.
- Binary tree is represented by its Root node. So a pointer must always be storing the root node address, which should not be lost.

Traversal in Binary Tree

- There are three types of traversals in Binary tree :
- Preorder Traversal : Root -> Left Tree -> Right Tree
- Inorder Traversal : Left Tree -> Root -> Right Tree
- Postorder Traversal: Left Tree -> Right Tree -> Root

Height of Binary Tree

- If the pointer is null, then return 0 (Height of empty tree is 0).
- Height of a tree is equal to 1 + the height of left or right sub tree, which ever is more.
- This can be implemented recursively by calculating left and right subtree heights by calling itself.
- Then compare them, and return the biggest of both.

Binary Search Tree

- One of the important uses of binary trees is searching. Every node is associated with a search key in binary search tree.
- A binary search tree satisfies the below rules :
  - All the keys in the left subtree of root are less than the key in the root.
  - All the keys in the right subtree of root are greater than the key in the root.
  - Left and right subtrees of root are also binary search trees.

Searching in binary search tree

- Start at the root node and move down.
- While descending, whenever we encounter a node, we have to compare the desired key with the key of that node.
  - If the desired key is equal to the key in the node, then search is successful.
  - If the desired key is less than the key of the node, then move to the left child.
  - If the desired key is greater than the key of the node, then move to the right child.
- Repeat the process until the search is successful or we reach a NULL.

Insertion at the end

- Start at the root node and move down.
- While descending, whenever we encounter a node, we have to compare the key to be inserted with the key of that node.
  - If the key to be inserted is equal then there is nothing to be done as Binary search tree does not allow duplicate keys.
  - If the key to be inserted is less than the key of the node, then move to the left child.
  - If the key to be inserted is greater than the key of the node, then move to the right child.

Repeat the process until we reach a parent with a NULL left or right child where the key to be inserted can fit.

Deletion
when node is a leaf node

- When a leaf node is to be deleted, the corresponding link node from the parent is made null, and the leaf node is released.

Deletion
when node has exactly one child

- When the node to be deleted has one child, then its child will take its place, in the parent's link field.

KERNEL MASTERS

Deletion
when node has exactly two children.

- We have to find the inorder successor of the node. Data from the inorder successor is copied to the node and then the inorder successor is deleted from the tree, using one of the first 2 methods, as the inorder successor will only have one child or no child.
- Inorder successor of a node is the node that comes just after that node in the inorder traversal of the tree.
- To find inorder successor of a node, we move to the right child of the node to be deleted, and keep on moving left until we find a node with no left child.