ADS answers

a. Sorting definition in general.

Sorting is any process of arranging items systematically, and has two common, yet distinct meanings:

1. ordering: arranging items in a sequence ordered by some criterion;
2. categorizing: grouping items with similar properties.

---

b. Bubble Sort

i. Idea of how it works.

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

So we will have 2 loops:

The first loop will start from 0 to n (where n is the length of our list) and then we will have our second loop which will also starts from 0 to n the we will have a condition that will check if the number in the array of the index in the second loop and compare the number with the next number and if the next number is smaller than the current number we will swap the number and we will keep going till the inner loop ends then we will continue with the outer loop so we will have a sorted list by the end:

First Pass

(**5 1** 4 2 8) → (**1 5** 4 2 8), Here algorithm compares the first two elements and swaps since 5 > 1.
(1 **5 4** 2 8) → (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) → (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) → (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass

(**1 4** 2 5 8) → (**1 4** 2 5 8)
(1 **4 2** 5 8) → (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) → (1 2 **4 5** 8)
(1 2 4 **5 8**) → (1 2 4 **5 8**)

Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass

(**1 2** 4 5 8) → (**1 2** 4 5 8)
(1 **2 4** 5 8) → (1 **2 4** 5 8)
(1 2 **4 5** 8) → (1 2 **4 5** 8)
(1 2 4 **5 8**) → (1 2 4 **5 8**)

ii. Complexity of the algorithm (worst case).

Worst and Average Case Time Complexity: O(n*n). Worst case occurs when array is reverse sorted.
Best Case Time Complexity: O(n). Best case occurs when array is already sorted.
Auxiliary Space: O(1)

iii. If it's an in-place/requires additional data structure kind of algorithm.

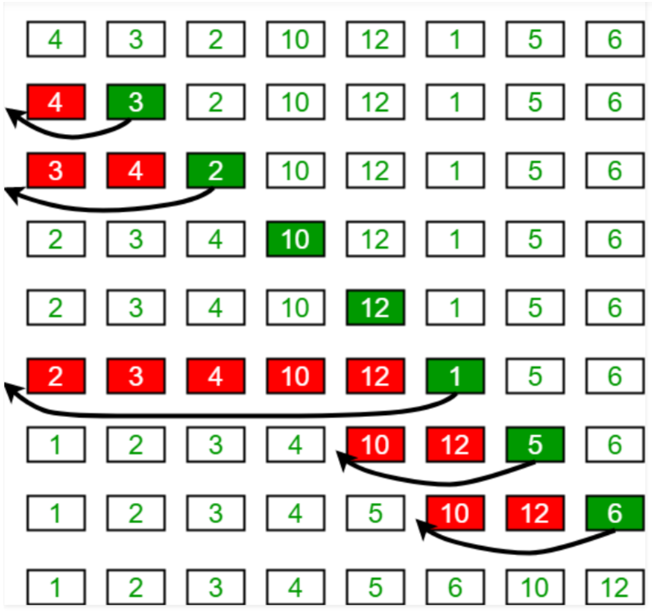
Yes it is an in-place algorithm since it requires some additional data (aux variables) for swapping.

---

c. Insertion Sort

i. Idea of how it works.

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands and it work in the following way:
To sort an array of size n in ascending order:
1: Iterate from arr[1] to arr[n] over the array.
2: Compare the current element (key) to its predecessor.
3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.



ii. Complexity of the algorithm (worst case).

Worst case time complexity: **O(n^2)**

Average case time complexity: **O(n^2)**

Best case time complexity: **O(n)**

Space complexity: **O(1)**

iii. If it’s an in-place/requires additional data structure kind of algorithm.

Yes it is an in-place sorting Algorithm because it requires some additional data struct (aux variable) to store the key.

---

d. Selection Sort

i. Idea of how it works.

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. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

```
 arr[] = 64 25 12 22 11


// Find the minimum element in arr[0...4]

// and place it at beginning

11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

ii. Complexity of the algorithm (worst case)

Time Complexity: O(n^2) in all of the cases as there are two nested loops.

iii. If it’s an in-place/requires additional data structure kind of algorithm

Yes it is an in-inplace sorting algorithm since there is some additional data (aux variables) for swapping and choosing the minimum element .

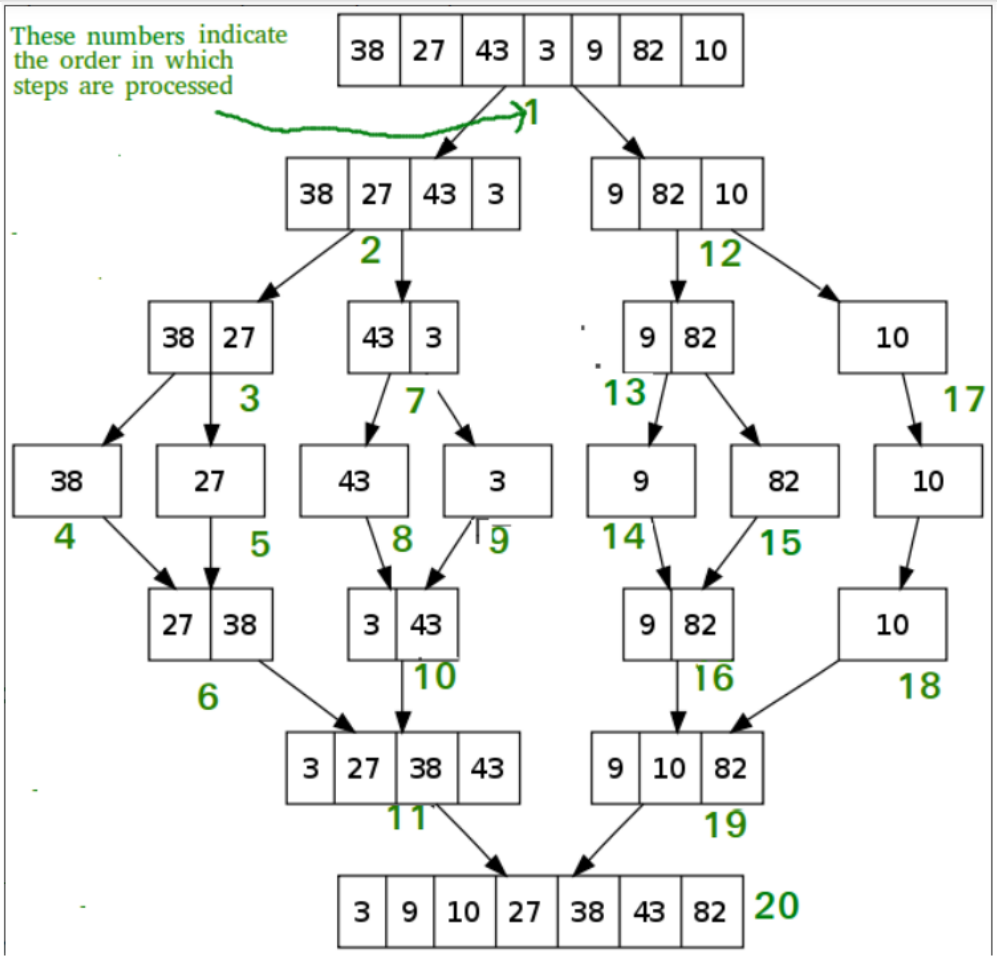
---

e. Merge Sort

i. Idea of how it works (General Pattern).

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

```
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)
```



ii. Complexity of the algorithm (worst case).

Time complexity of Merge Sort is θ(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

iii. If it’s an in-place/requires additional data structure kind of algorithm.

The standard merge sort on an array is not an in-place algorithm, since it requires O(N) additional space to perform the merge.

iv. Merge and its complexity

The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See the following C implementation for details.

```
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r){
    int n1 = m - l + 1;
    int n2 = r - m;
    // Create temp arrays
    int L[n1], R[n2];
    // Copy data to temp arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    // Merge the temp arrays back into arr[l..r]
    // Initial index of first subarray
    int i = 0;
    // Initial index of second subarray
    int j = 0;
    // Initial index of merged subarray
    int k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;}
        else {
            arr[k] = R[j];
            j++;}
        k++;}
    // Copy the remaining elements of
    // L[], if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++; k++;}
    // Copy the remaining elements of
    // R[], if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

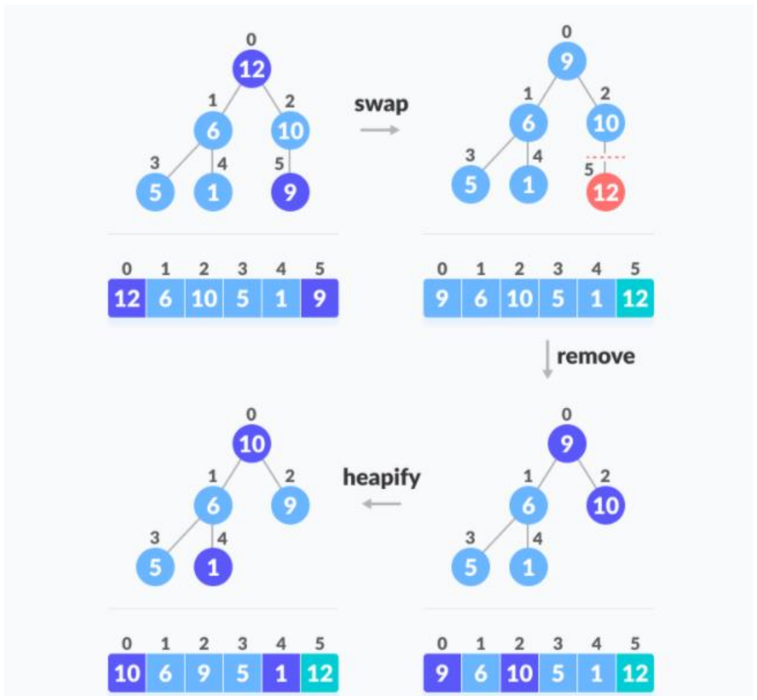
v. Brief explanation of why/how sentinel is used.

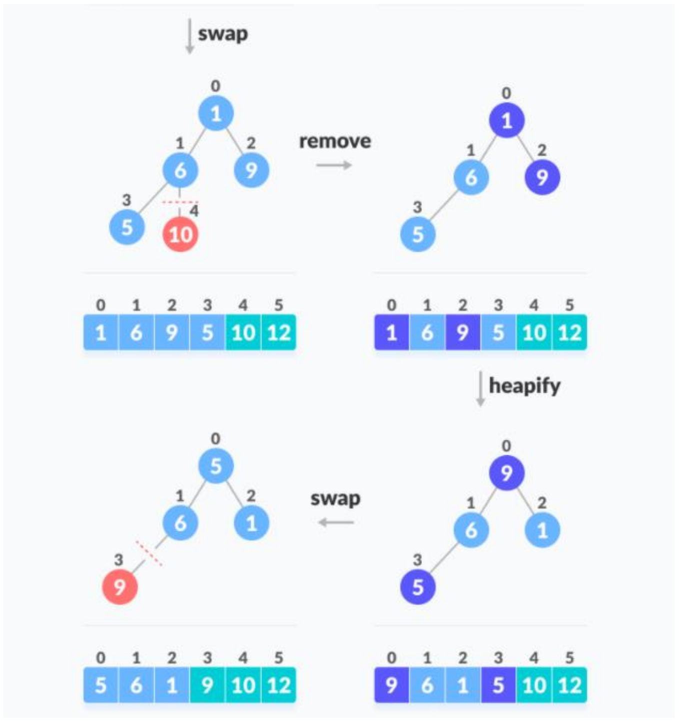
f. Heap Sort

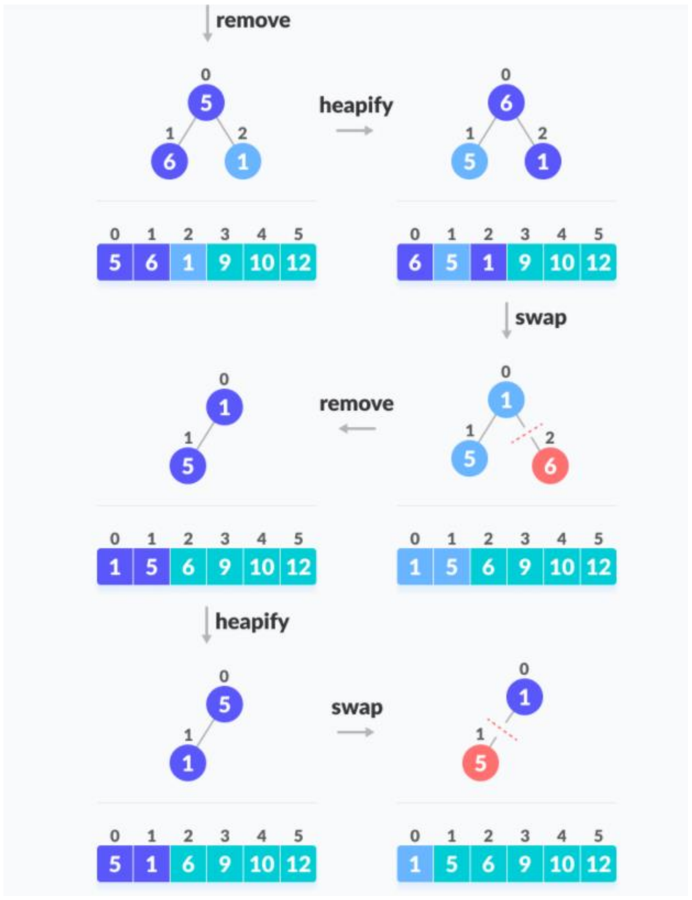
i. Idea of how it works (General Pattern).

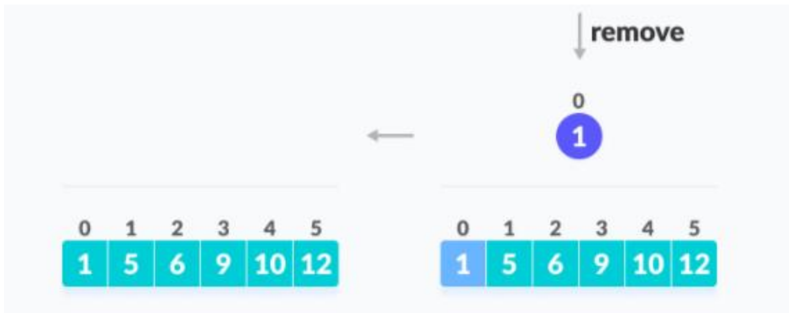
Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for the remaining elements:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.
3. Repeat step 2 while size of heap is greater than 1.




swap
remove
heapify
swap


remove
heapify
swap
remove
heapify
swap



ii. Complexity of the algorithm (worst case).

Time complexity of heapify is O(Log n). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(n Log n).

iii. If it's an in-place/requires additional data structure kind of algorithm.

Yes it is an in-place algorithm because each operation of the heap sort algorithm requires a number of non-array variables for temporary storage, loop indexing, and the like that does not depend on the size of the input array.

iv. How is max selection sort related to heap sort. Brief explanation of how it works

Thus, to maintain the max-heap property in a tree where both sub-trees are max-heaps, we need to run heapify on the root repeatedly element until it is larger than its children or it becomes a leaf node , and here where we need to do the max selection sort where we choose the max element in the tree and we put it in the main node then we cut it to but it at the end of our list.

v. Preprocessing related in heap sort

vi. Resorting heap property

In a heap, for every node *i* other than the root, the value of a node is greater than or equal (at most) to the value of its parent.
Thus, the largest element in a heap is stored at the root.

vii. Tree property related to building a heap.

The tree property in the heap building should be a complete binary search tree , A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

viii. What is heapify? What is its complexity?

It is the method that we use to build our binary Heap and it works like the following :

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as an array and the array-based representation is space-efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

```
Input data: 4, 10, 3, 5, 1
         4(0)
        /   \
     10(1)   3(2)
    /   \
 5(3)    1(4)

The numbers in bracket represent the indices in the array
representation of data.

Applying heapify procedure to index 1:
         4(0)
        /   \
    10(1)    3(2)
    /   \
5(3)    1(4)

Applying heapify procedure to index 0:
        10(0)
        /  \
     5(1)  3(2)
    /   \
 4(3)    1(4)
The heapify procedure calls itself recursively to build heap
 in top down manner.
```

Time complexity of heapify is O(Log n)

ix. Why is heap sort an asymptotically sorting algorithm?

Heap sort is an asymptotically sorting algorithm because it is a comparison-based sorting algorithm. A comparison-based sorting algorithm is a sorting algorithm whose only access to the data is a comparison oracle, which compares to data items. Any such algorithm must make $\Omega(n \log n)$ oracle queries in the worst case. Therefore a comparison-based sorting algorithm running in time $O(n \log n)$ is asymptotically optimal.

g. Quick Sort

i. Idea of how it works (General Pattern).

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

ii. Complexity of the algorithm (worst case and best case).

Worst Case: O(n^2)

Best Case: O(n Log n)

Average Case: O(n Log n)

iii. If it's an in-place/requires additional data structure kind of algorithm.

Yes it is an in-place sorting algorithm since it doesn't require and additional data structure.

iv. Explain partitioning in quick sort.

```
/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

h. Quick Select

i. Idea of how it works (General Pattern).

Quickselect is a selection algorithm to find the k-th smallest(or biggest) element in an un-ordered list.

The algorithm is similar to QuickSort. The difference is, instead of recurring for both sides (after finding pivot), it recurs only for the part that contains the k-th smallest element. The logic is simple, if index of partitioned element is more than k, then we recur for left part. If index is same as k, we have found the k-th smallest element and we return. If index is less than k, then we recur for right part

ii. Complexity of the algorithm (worst case and best case).

Beat case is O(n) and worst case is O(n^2).

iii. If it's an in-place/requires additional data structure kind of algorithm.

Yes it is.

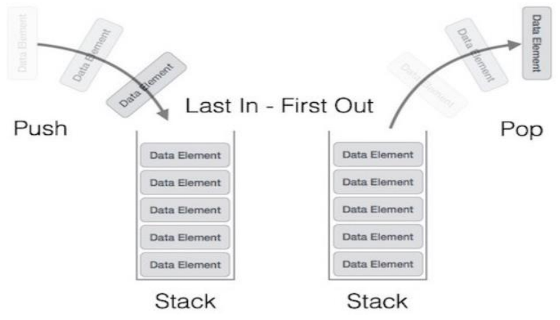
---

i. Stack

i. General explanation of the data structure.

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

ii. Operations used on stack: Push, Pop, Empty (have to mention how they work/their concept)



Push( ): The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.

- **Step 5** − Returns success.

```
•  begin procedure push: stack, data
•
•     if stack is full
•        return null
•     endif
•
•     top ← top + 1
•     stack[top] ← data
•
•  end procedure
```

Pop( ): Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.

```
•  begin procedure pop: stack
•
•     if stack is empty
•        return null
•     endif
•
•     data ← stack[top]
•     top ← top - 1
•     return data
•
•  end procedure
```

isempty( ): Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty.

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif

end procedure
```

iii. Implementation of the stack handling the operations and empty case

peek() − get the top data element of the stack, without removing it.

```
begin procedure peek
return stack[top]
end procedure
```

if the stake is empty we will return null.

isFull() − check if stack is full.

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

---

j. Queue

i. General explanation of the data structure.

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology

ii. Operations used on queue: Enqueue, Dequeue (have to mention how they work/their concept)

Enqueue

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.

- **Step 5** − return success.

```
procedure enqueue(data)

   if queue is full
      return overflow
   endif

   rear ← rear + 1
   queue[rear] ← data
   return true

end procedure
```

Dequeue

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.

```
procedure dequeue

   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1
   return true

end procedure
```

iii. Implementation of the stack handling the operations and empty case

During the enqueue and the dequeue if the queue is empty we will return some kind of underflow .

- **peek()** − Gets the element at the front of the queue without removing it.

```
begin procedure peek
   return queue[front]
end procedure
```

- **isfull()** − Checks if the queue is full.

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

- **isempty()** − Checks if the queue is empty.

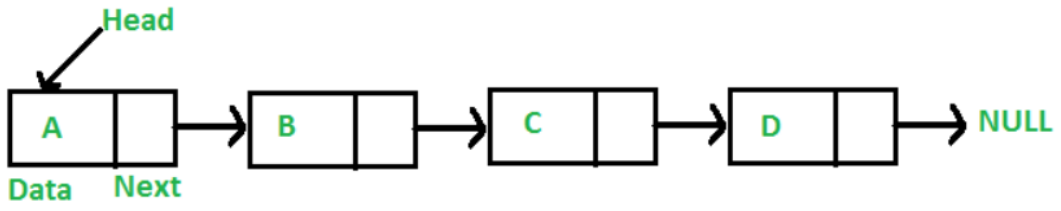
```
 begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

---

k. Linked Lists

i. General explanation of the data structure.

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:

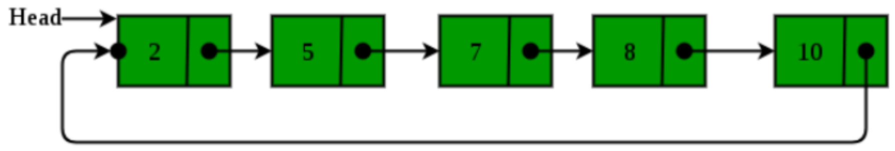


In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list

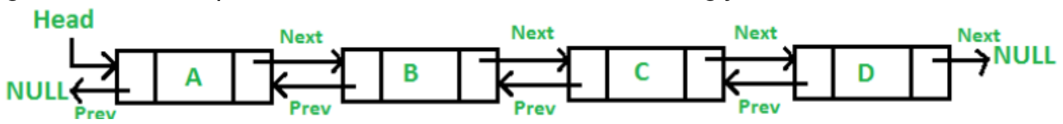
ii. Types of linked lists with their description

- Singly Linked List
  It has a pointer that points to the next element only.
- Circular Linked List

is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



- Doubly Linked List
  A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



iii. Operations used on linked lists: Search, Insertion, Delete (have to mention how they work/their concept and their complexities)

Search:

Write a function that searches a given key ‘x’ in a given singly linked list. The function should return true if x is present in linked list and false otherwise.

```
   bool search(Node *head, int x)
```

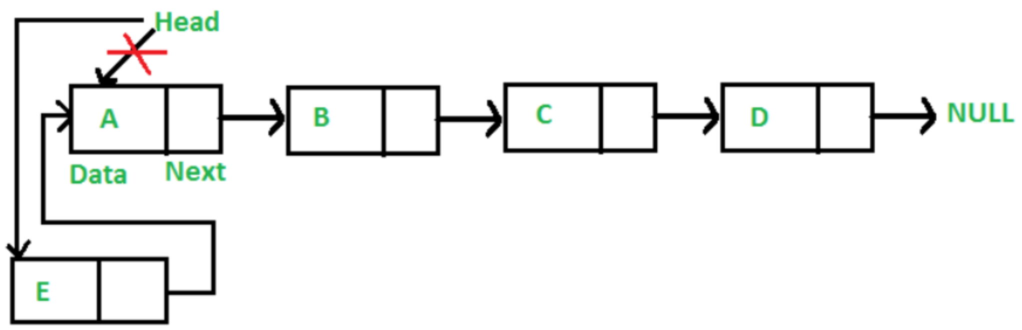
For example, if the key to be searched is 15 and linked list is 14->21->11->30->10, then function should return false. If key to be searched is 14, then the function should return true.

Iterative Solution

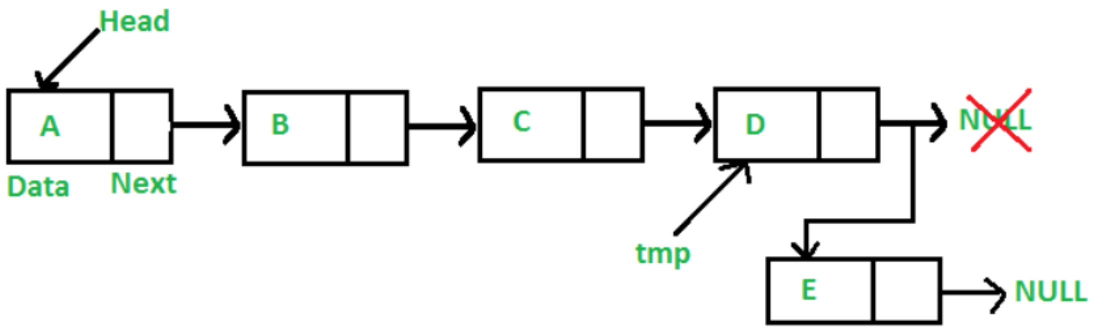
```
2) Initialize a node pointer, current = head.
3) Do following while current is not NULL
    a) current->key is equal to the key being searched return true.
    b) current = current->next
4) Return false
```

Insertion:

Add a node at the front: (4 steps process)
The new node is always added before the head of the given Linked List. And newly added node becomes the new head of the Linked List. For example if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node

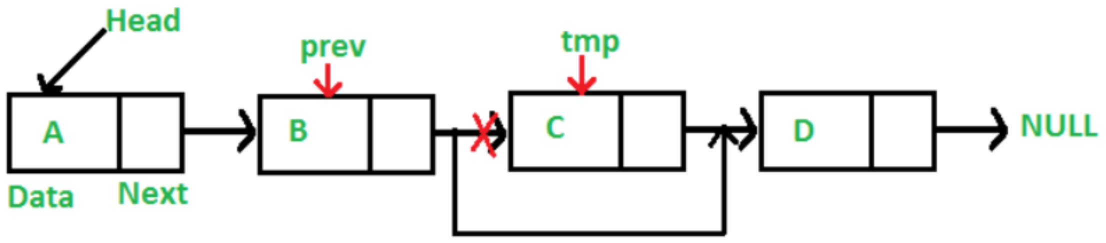


Add a node at the end: (6 steps process)
The new node is always added after the last node of the given Linked List. For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.
Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.



Delete:

To delete a node from the linked list, we need to do the following steps.
1) Find the previous node of the node to be deleted.
2) Change the next of the previous node.
3) Free memory for the node to be deleted.



iv. Implementation of the stack handling the operations and empty case

v. Double Linked lists with sentinel: General idea -> with empty -> Search -> Insert -> Delete

I. Binary Search Trees

i. General explanation of the data structure.

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

- The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

```
left_subtree (keys) < node (key) ≤ right_subtree (keys)
```

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

ii. Pointers used with it

binary search trees are also **pointer** based data structures, meaning they don't have an upper bound on storage (aside from the physical limits of hardware). They can be grown or shrunk indefinitely, as they simply allocate memory for a new node when they need to grow.

iii. Its property

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

iv. Traversals: General idea of how each traversal works with BST (no need for mentioning example if concept of each is properly explained)

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees:

(a) Inorder (Left, Root, Right) :

```
Algorithm Inorder(tree)

   1. Traverse the left subtree, i.e., call Inorder(left-subtree)

   2. Visit the root.

   3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```

(b) Preorder (Root, Left, Right) :

```
Algorithm Preorder(tree)

   1. Visit the root.

   2. Traverse the left subtree, i.e., call Preorder(left-subtree)

   3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

(c) Postorder (Left, Right, Root) :

```
Algorithm Postorder(tree)

   1. Traverse the left subtree, i.e., call Postorder(left-subtree)

   2. Traverse the right subtree, i.e.,call Postorder(right-subtree)

   3. Visit the root.
```

v. Operations for BST: Search, Min, Max, Previous, Next, Insert, Delete (have to mention how they work/their concept and their complexities)

a) Search: Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree
Time Complexity : O(N).

b)Insert: Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.
Time Complexity : O(N).

c)Min: This is quite simple. Just traverse the node from root to left recursively until left is NULL. The node whose left is NULL is the node with minimum value.
Time Complexity: O(N).

d)Max: This is quite simple. Just traverse the node from root to right recursively until right is NULL. The node whose right is NULL is the node with maximum value.
Time Complexity: O(N).

e)Deletion: When we delete a node, three possibilities arise.

1) ***Node to be deleted is leaf:*** Simply remove from the tree.

```
             50                               50
          /     \         delete(20)      /   \
         30      70       --------->    30     70
        /  \    /  \                     \    /  \
      20   40  60   80                   40  60   80
```

2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child

```
             50                              50
          /     \         delete(30)      /   \
         30      70       --------->    40     70
           \    /  \                          /  \
           40  60   80                      60   80
```

3) ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

```
             50                              60
          /     \         delete(50)      /   \
         40      70       --------->    40    70
                /  \                            \
              60   80                           80
```

The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

vi. Know the best and worst shapes for BST

The binary search tree is a skewed binary search tree.

The binary search tree is a balanced (complete) binary search tree.

m. Priority Queue

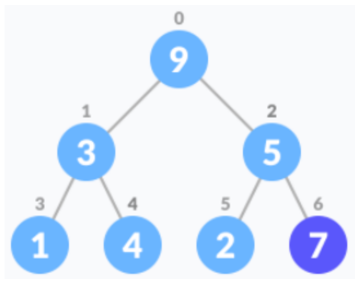
i. General explanation of the data structure.

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority.

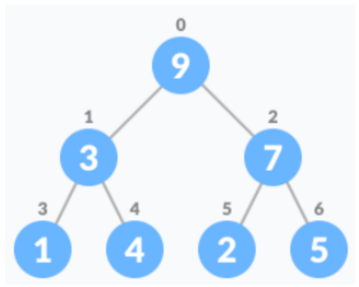
ii. Operations for priority queue: insertion, deletion (with complexities)

1. **Inserting an Element into the Priority Queue**

Inserting an element into a priority queue (max-heap) is done by the following steps.Insert the new element at the end of the tree.



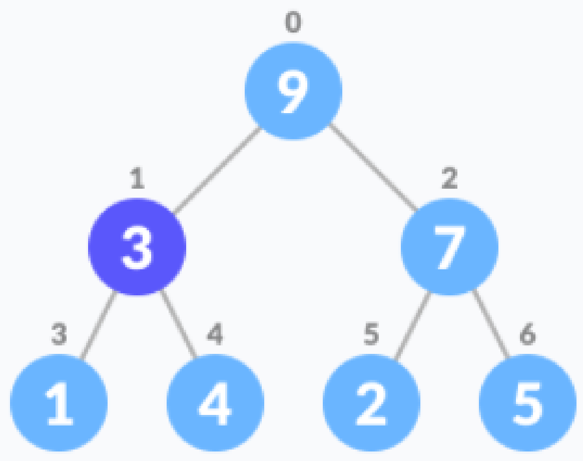
then we Heapify



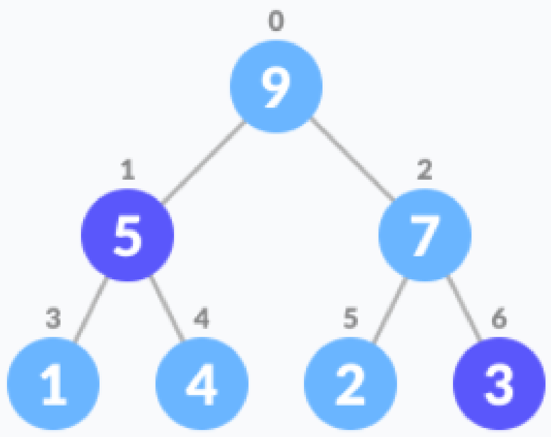
2. Deleting an Element from the Priority Queue

Deleting an element from a priority queue (max-heap) is done as follows:

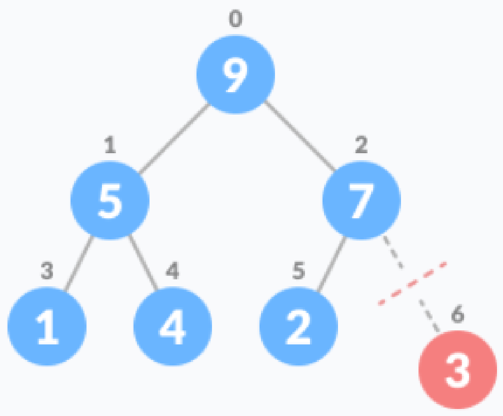
Select the element to be deleted.



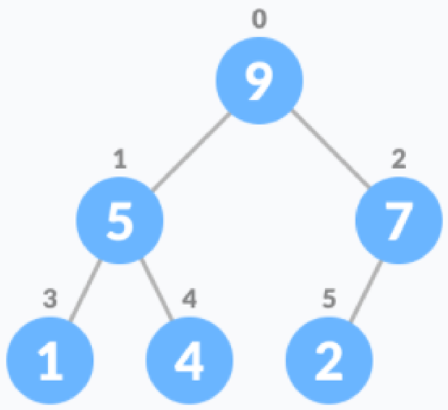
Swap it with the last element.



Remove the last element.



Heapify the tree.



The time complexity of all the operation is **O(log N)**

iii. Explain trick on how to delete the maximum element

If it is a max-heap we choose the main root then we delete it and if it is a min-heap we just remove the last element. (I am not sure)

---

n. Hashing

i. General explanation

Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the hash function used.
# Hash function : It is a special function that we can get the index of the value by using the key and we have 3 types for this :

a) For numbers we can do this function:
   ((the number) mod (the number of the places))
b) For words we can get the ASCII code of each litter and mod the number of the empty places : ((sum of ASCII codes) mod (the number of the places))
c) For big number "like 2345678910" we can divide the numbers into 2 parts and then add them and then we got mod the number of the places :
   (23 + 45 + 67 + 89 + 10) mod (the number of the places)

ii. Chaining

1.General explanation

2.Operations used with it and their complexities

Answer for 1 and 2:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple , but requires additional memory outside the table.

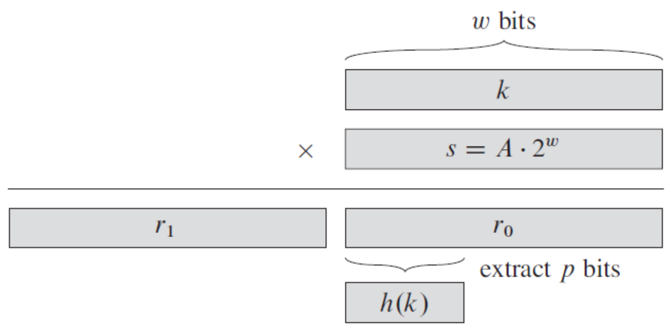
3. Division Method

In the ***division method*** for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

$h(k) = k \bmod m$

5. Multiplication Method

The ***multiplication method*** for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the



iii. Opening Addressing:

1. General explanation
2. Operations: Insert, Search, Delete.

Answer for 1 and 2:

In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

iv. Linear Hashing: General Explanation

We might get that 2 keys have the same index in our list and we call this Collisions and in this case we check the next place if it is empty we put our key on it … if it is not empty we check the next index , if not we keep checking until we get a free place to put our key , this method is called linear probing.

v. Quadratic Hashing: General Explanation

We might get that 2 keys have the same index in our list and then this is a t Collisions, in this case we check the next 1^2 place if it is empty we put our key on it … if it is not empty we check the 2^2 index , if not we keep checking quadraticly until we get a free place to put our key , we also need to make sure that our index won't be our of the rang of our list so we need to move back over to the beginning of the list to make sure that we are in the right way.

vi. Double Hashing: General Explanation

we need to have here 2 hash code function (we can use the Division method and the Multiplicition method to get 2 hash functions) then we need to make sure that :

a) The 2 hash code function must have a different results
b) Non of them return NULL

We apply the function "hashcode()" to the variable 'x' the if we applied the same function to the variable 'r' we need to apply the function "hashcode2()" for 'r' and then we add the results and we get our index or key