# 1no.

Here is NodeType is a structure that has

```
struct NodeType{
int info;
NodeType* next;
};
//listData is a pointer of type NodeType that will hold the
//address the LinkedList
void DeleteItem(int item) {
   NodeType *location = listData;
   NodeType *tempLocation=NULL;
   if (item == listData->info) {
      tempLocation = location;
      listData = listData->next;
      delete tempLocation;
      length--;
      return;

   }
   while((item!=(location->next)->info) && (location->next != NULL)){
```

```
        location = location->next;
    }
    if(location->next != NULL){
        tempLocation = location->next;
        location->next = (location->next)->next;
        delete tempLocation;
        length--;
        return;
    }
}
```

---

```
void RetrieveItem(int &item, bool &found) {
    NodeType *location = listData;
    bool moreToSearch = (location != NULL);
    found = false;
    while (moreToSearch && !found) {
        if (item == location->info)
            found = true;
        else if (item > location->info) {
            location = location->next;
            moreToSearch = (location != NULL);
```

```
        } else
            moreToSearch = false;
    }
}
```

Time complexity:

DeleteItem(int item) : Here the time complexity is depends on the run time of while loop. Because the other statement has the time complexity of 1.

The while loop will iterate the whole list until it finds the item. If the size of list is n, then we can represent the time complexity in the notation of Big-O: O(n).

RetrieveItem(int &item, bool &found): Here also the while loop will iterate the whole list until it finds the desire item. If the size of the list is n then we can represents the time complexity in the notation of Big-O: O(n).

## 2no.

If we don't use the tail pointer then we can't traced the next item since we will not have the address of the next item. So to hold the next item's address we need a tail pointer to hold the address.

## 3no.

```
//Header file:
#ifndef SORTEDTYPE_H_INCLUDED
#define SORTEDTYPE_H_INCLUDED
template<class T>
class SortedType{
  struct NodeType{
    T info;
    NodeType* next;

  };

  public:
     SortedType();
     ~SortedType();
     int LengthIs();
     bool IsFull();
     void MakeEmpty();
     void InsertItem(T);
     void RetrieveItem(T&, bool&);
```

```
      void DeleteItem(T);
      void ResetList();
      void GetNextItem(T&);
      //friend void print(timeStamp&);
   private:
    NodeType* listData;
    int length;
    NodeType* currentPos;

};
#endif // SORTEDTYPE_H_INCLUDED

//source file:
#include <bits/stdc++.h>
#include "sortedtype.h"
using namespace std;

template<class ItemType>
SortedType<ItemType>::SortedType() {
    length = 0;
    listData = NULL;
```

```
    currentPos = NULL;
}

template<class ItemType>
int SortedType<ItemType>::LengthIs() {
    return length;
}

template<class ItemType>
bool SortedType<ItemType>::IsFull() {
    NodeType *location;
    try {
        location = new NodeType;
        delete location;
        return false;
    }
    catch (bad_alloc &exception) {
        return true;
    }
}
```

```
template<class ItemType>
void SortedType<ItemType>::InsertItem(ItemType item) {
    NodeType *newNode;
    NodeType *predLoc;
    NodeType *location;
    bool moreToSearch;
    location = listData;
    predLoc = NULL;
    moreToSearch = (location != NULL);
    while (moreToSearch) {
        if (location->info < item) {
            predLoc = location;
            location = location->next;
            moreToSearch = (location != NULL);
        } else
            moreToSearch = false;
    }
    newNode = new NodeType;
    newNode->info = item;
    if (predLoc == NULL) {
        newNode->next = listData;
```

```
        listData = newNode;
    } else {
        newNode->next = location;
        predLoc->next = newNode;
    }
    NodeType* temp = listData;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = listData;
    length++;
    delete temp;
}

template<class ItemType>
void SortedType<ItemType>::DeleteItem(ItemType item) {
    NodeType *location = listData;
    NodeType *tempLocation=NULL;
    if (item == listData->info) {
        tempLocation = location;
        listData = listData->next;
```

```
        delete tempLocation;
        length--;
        NodeType* temp = listData;
        while (temp->next != nullptr) {
        temp = temp->next;
    }
      temp->next = listData;
      length++;
      delete temp;
      return;
 }
   while((item!=(location->next)->info) && (location->next != NULL)){
         location = location->next;
    }
   if(location->next != NULL){
        tempLocation = location->next;
        location->next = (location->next)->next;
        delete tempLocation;
        length--;
        NodeType* temp = listData;
```

```
        while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = listData;
    length++;
    delete temp;

        return;
    }
}

template<class ItemType>
void SortedType<ItemType>::RetrieveItem(ItemType &item,
bool &found) {
    NodeType *location = listData;
    bool moreToSearch = (location != NULL);
    found = false;
    while (moreToSearch && !found) {
        if (item == location->info)
            found = true;
        else if (item > location->info) {
```

```
            location = location->next;
            moreToSearch = (location != NULL);
        } else
            moreToSearch = false;
    }
}

template<class ItemType>
void SortedType<ItemType>::MakeEmpty() {
    NodeType *tempPtr;
    while (listData != NULL) {
        tempPtr = listData;
        listData = listData->next;
        delete tempPtr;
    }
    length = 0;
}

template<class ItemType>
SortedType<ItemType>::~SortedType() {
    MakeEmpty();
```

```
}

template<class ItemType>
void SortedType<ItemType>::ResetList() {
   currentPos = NULL;
}

template<class ItemType>
void SortedType<ItemType>::GetNextItem(ItemType &item) {
   if (currentPos == NULL)
      currentPos = listData;
   else
      currentPos = currentPos->next;
   item = currentPos->info;
}
```

## 4no.

The snippet of a code is about the implementation of Queue using array .

We conventionally, indexed the first element of the array is 0. By convention Queue is follow the first in first out. So we insert

the element in the Queue by the calculation of rear = ( rear + 1) %maxQue, which give me the index of the array by assuring that the index is the range of the array size.

Also this modulo operator wrapping around the Queue. When Dequeue a Queue it comes up front = rear = 0 index.

# 5no.

Stack: A Stack is a list of elements in which an element may be inserted or deleted only at one end, call top of the Stack. It follow the last in first out. Meaning if an element enter last in a stack then it out very first and first inserted element will pop very last.

In a text editor, stack is used for undo features.

In programming, a stack is used for managing function calls and local variables during the execution of a program.

Queue: A Queue is a data structure that follow the first in first out exactly like peoples are waiting in front of ticket counter in a queue, the first one will get the first ticket and will go out from the queue.

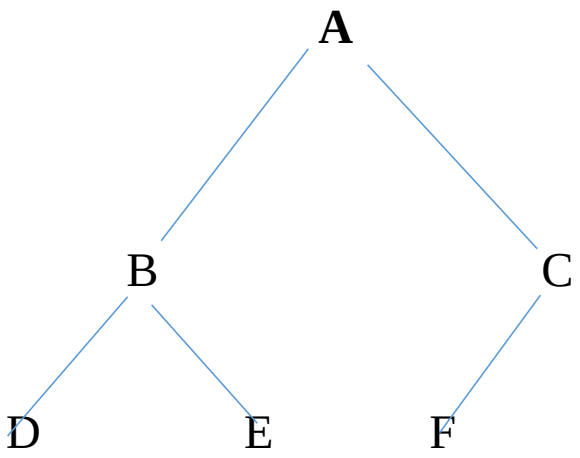
When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling

Queue is used when things don't have to be processed immediately, but have to be processed in First In First Out order

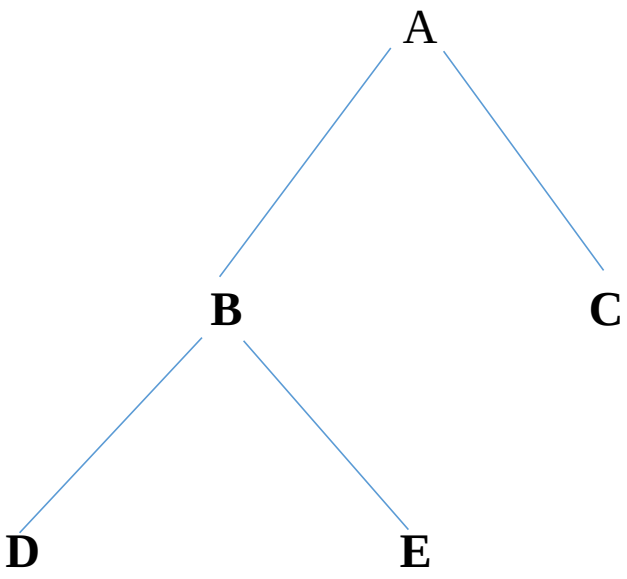
like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

# 6no.

A complete binary tree is a tree at every level of the tree is completely filled and if the last level is not filled, then it will filled from the most left.




A full binary tree is a tree at every level of the tree either a node has a zero child node or two child node.

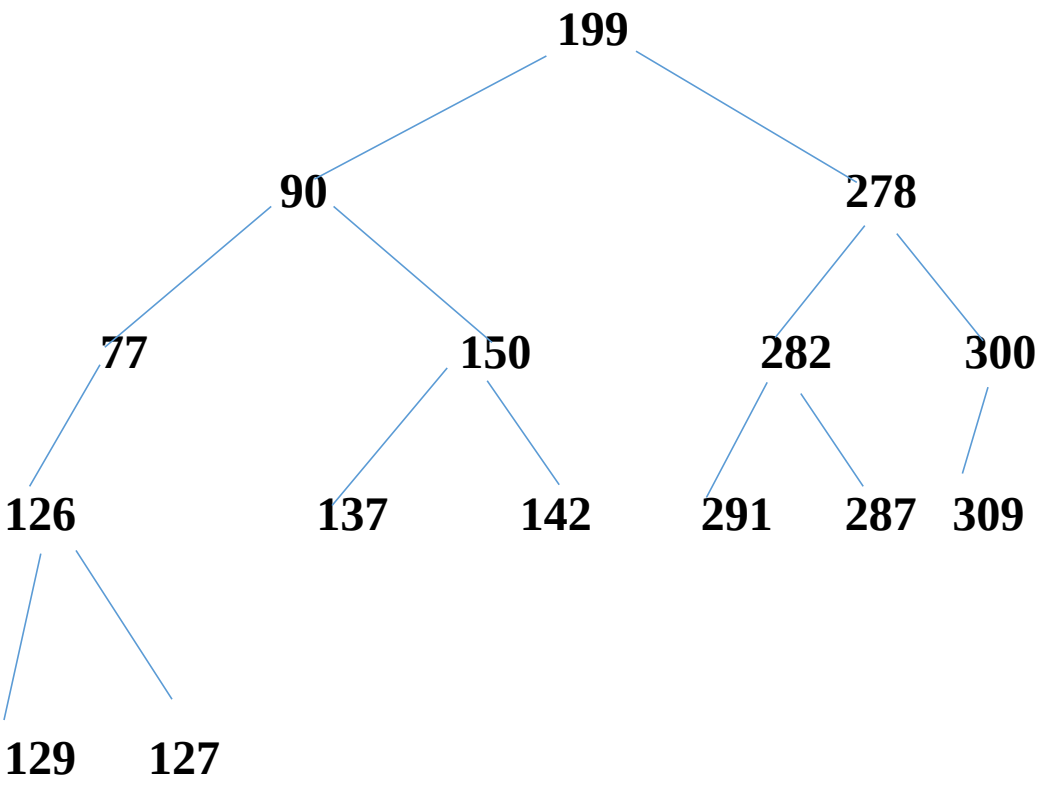



# 7no.

If the number of nodes n then the formula of height h is

$h = \log_2(n+1)$, at where we will take the ceiling value .

# 8no.




**Post order :**129, 127, 126, 77, 137, 142, 150,90, 291, 287, 282, 300, 309, 278, 199