Lab No.1 Introduction to Pointers and Arrays

OBJECTIVES OF THE LAB

In this lab, we will learn about the relation between arrays and pointers, and use them efficiently in our program.

- *Pointers and dynamic memory allocation*
- *Arrays*
- *Connection between arrays and pointers*

1.1 Pre Lab

1.1.1 Pointers

A C pointer is a variable (memory storage) large enough to accommodate the value of a memory address. C pointer simply holds an integer (not the actual memory object it may be pointing to). For pointers to be useful there need to be some other memory objects that the pointers can point to.

Example:

```
1) int i = 7;              /* simple integer variable */

2) int *ptr = NULL;       /* simply holds the value of NULL */

3) ptr = i;               /* bad for your health, but legal */

4) ptr = &i;              /* takes address of i */

5) *ptr = 8;              /* dereference the pointer to access i i.e. the value
                          referenced to by ptr is equal to 8* means assign 8 to the
                          address of this pointer ie at i/
```

1.1.2 Initializing Pointers

Pointers can be initialized to point to specific locations at the very moment they are defined:

```
int myvar;
int * myptr = &myvar;
```

The resulting state of variables after this code is the same as after:

```
int myvar;
int * myptr;
myptr = &myvar;
```

When pointers are initialized, what is initialized is the address they point to (i.e., myptr), never the value being pointed (i.e., *myptr). Therefore, the code above shall not be confused with:

```
int myvar;
int * myptr;
*myptr = &myvar;
```

Which anyway would not make much sense (and is not valid code).
The asterisk (*) in the pointer declaration (line 2) only indicates that it is a pointer, it is not the dereference operator (as in line 3). Both things just happen to use the same sign: *. As always, spaces are not relevant, and never change the meaning of an expression.

Pointers can be initialized either to the address of a variable (such as in the case above), or to the value of another pointer (or array):

```
int myvar;
int *myptr = &myvar;
int *bar = myptr;
```

Example:

```
// my first pointer
#include <iostream>
using namespace std;

int main ()
{
  int first, second;
  int *mytr;

  myptr = &first;
  *myptr = 10;
  myptr = &second;
  *myptr = 20;
  cout << "firstvalue is " <<
first << '\n';
  cout << "secondvalue is " <<
second << '\n';
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

1.1.3 Dynamic Memory Allocation (malloc function/operator new, delete)

In most programs all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators new and delete. The syntax is:

```
pointer = new type
pointer = new type [number_of_elements]
delete pointer;
delete[] pointer;
```

C++ integrates the operators new and delete for allocating dynamic memory. But these were not available in the C language; instead, it used a library solution, with the functions malloc, calloc, realloc and free, defined in the header <cstdlib> (known as <stdlib.h> in C). The functions are also available in C++ and can also be used to allocate and deallocate dynamic memory.

Note, though, that the memory blocks allocated by these functions are not necessarily compatible with those returned by new, so they should not be mixed; each one should be handled with its own set of functions or operators.

- The size of the problem often cannot be determined at "compile time".
- Dynamic memory allocation is to allocate memory at "run time".
- Dynamically allocated memory must be referred to by pointers.

Example:

```
int array_size=100;

double a[100], *b, *c;

b = (double *) malloc(array_size * sizeof(double));          /* allocation in C*/

c = new double[array_size];                          /* allocation in C++ */
```

1.1.4 Pointers and arrays

The concept of arrays is related to that of pointers. In fact, arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type. For example, consider these two declarations:

```
int a[100], *ptr_a;     /*a is an array of 100 elements and ptr_a is a pointer*/

ptr_a = &(a[0]);        /* or ptr_a = a;  a pointer pointing to the first location of array*/

ptr_a++;                /*or ptr_a += 1; */


// now ptr_a points to the next integer, a[1];
```

After that, ***ptr_a*** and ***a*** would be equivalent and would have very similar properties. The main difference being that ***ptr_a*** can be assigned ***a*** different address, whereas a can never be assigned anything, and will always represent the same block of 100 elements of type int.

Example:

```
// Pointers and Array
#include <iostream>
using namespace std;

int main ()
{
  int MyArray[10];
```

```
100, 200, 300, 400, 500,
  int * ptr;
  ptr = MyArray;
  *ptr = 100;
  ptr++;
  *ptr = 200;
  ptr = &MyArray[2];
  *ptr = 300;
  ptr = MyArray + 3;
 *ptr = 400;
  ptr = MyArray;  *(ptr+4) = 500;
  for (int n=0; n<5; n++)
    cout << MyArray[n] << ", ";
  return 0;
}
```

1.2 Post Lab

Task 01

Write a program that reads numbers from the user in to an array of type "float", average them and print the result.

Sample OUTPUT of the program

Enter length of the array: 5

Enter the elements of the array: 11 22 33 44 55

Average of the array: 33

Note: Add all the elements of array and divide by total number of elements. Use pointers.

Task 02

Write a function that takes an int array and array's size as argument and return maximum value of array elements.

Prototype of function:

int maxValue(int arr[], int Size)

Call to the function:

int ReturnedValue = maxValue (passArrayHere,passSizeHere)

Note: Store first element of array in a temporary variable and then traverse the array and if you find any value greater than temp update temp. After traversal return temp which contains the maximum value.

Task 03

Write a function that takes an int array and the array's size as arguments. It should create a new array that is twice the size of the argument array. The function should copy the contents of the argument array to the new array, and initialize the unused elements of new array with -1. The function should return a pointer to the new array.

Note: Use malloc or new function from standard library for dynamic allocation of the array at runtime.

Task 04

Write a function that takes two int arrays and the arrays' sizes as arguments. It should create a new array big enough to store both arrays. Then it should copy the contents of the first array to the new array, and then copy the contents of the second array to the new array in the remaining elements, and return a pointer to the new array.

Prototype:

int* mergeArray (int arrA[], int sizeA, int arrB[], int SizeB)

Note: You can also use pointer to hold an array argument. So the above function can be replaced by the following function

Int * mergeArray(int* arrA, int sizeA, int* arrB, int sizeB)

1.3 References

1 **Introduction to Algorithms by *CLRS (3rd ed.)***

2 **Cplusplus.com**

Lab No.2 Sorting Algorithms Part (A)

OBJECTIVES OF THE LAB

In this lab, we will learn about some basic sorting techniques and algorithms.

- *Selection Sort*
- *Bubble Sort*
- *Insertion Sort*

2.1 Pre Lab

2.1.1 Sorting Algorithms

A sorting algorithm is an algorithm that puts elements of a list in a certain order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) which require input data to be in sorted lists; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

Input: A sequence of n numbers such as (a1, a2, …,an)
Output: A permutation (reordering) of the input sequence in sorted fashion. The permutation must either be in non-decreasing (or in some cases decreasing) order.

2.1.2 Bubble Sort [Best: O(N), Worst: O(N^2)]

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. The algorithm of Bubble sort is a quite simple. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm's average time and worst-case performance is O(n2), so it is rarely used to sort large, unordered data sets.

For example, if any number of elements is out of place by only one position (e.g. 0123546789 and 1032547698), bubble sort's exchange will get them in order on the first pass, the second pass will find all elements in order, so the sort will take only 2ntime.

Example:

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 our 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**)

2.1.3 Selection Sort [Best/ Worst: O(N^2)]

Selection sort is an in-place comparison sort. The algorithm finds the minimum element in the array (considering ascending order) from unsorted part and putting it at the beginning and repeats these steps for the remainder of the array. It does no more than n swaps, and thus is useful where swapping is very expensive. The algorithm maintains two sub-arrays in a given array.

1. The sorted sub-array.
2. The remaining unsorted sub-array.

In each iteration the minimum element from the unsorted sub-array is picked and moved to the sorted sub-array. It has O(n2) complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

Example:

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

2.1.4 Insertion Sort [Best: O(N), Worst: O(N^2)]

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hand. It is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and is often used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one.

Example:
12, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 5 (Size of input array)

i = 1.

Since 11 is smaller than 12, move 12 and insert 11 before 12
11, 12, 13, 5, 6

i = 2.

13 will remain at its position as all elements in A[0..I-1] are smaller than 13
11, 12, 13, 5, 6

i = 3.

5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.
5, 11, 12, 13, 6

i = 4.

6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
5, 6, 11, 12, 13

2.2 Post Lab

Task 01

Implement Selection Sort and analyze its worst, best and average case complexity.

Task 02

Implement Bubble Sort and analyze its worst, best and average case complexity.

Task 03

Implement Insertion Sort and analyze its worst, best and average case complexity.

2.3 References

1. **Introduction to Algorithms by *CLRS (3rd ed.)***
2. **Cplusplus.com**
3. **Class Notes**

Lab No.3 Sorting Algorithms (Part B)

OBJECTIVES OF THE LAB

In this lab, we will learn about some more sorting techniques and algorithms.

- *Merge Sort*
- *Quick Sort*

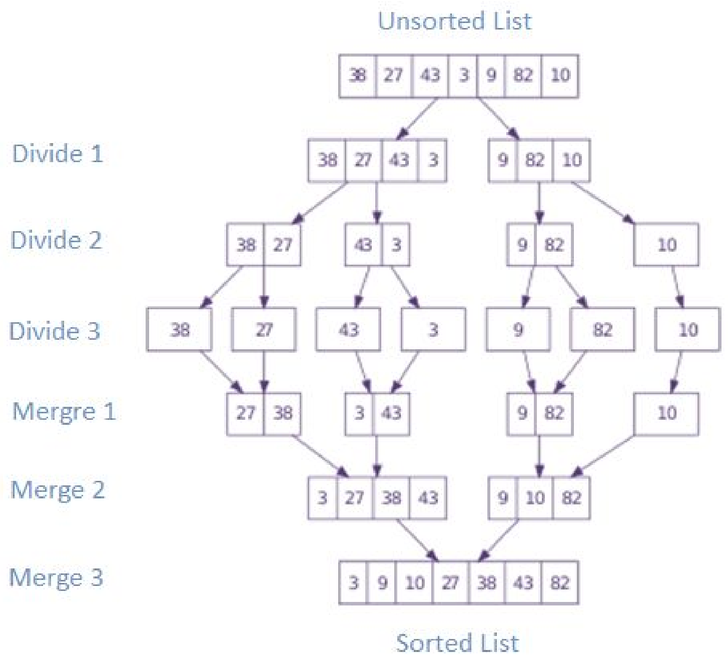
3.1 Pre Lab

3.1.1 Merge Sort [Best: O(n log n), Worst: O(n log n)]

Merge sort is an efficient, general-purpose, comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Merge sort is a divide and conquer algorithm which can easily be applied to lists, not only arrays, as it only requires sequential access, not random access. Conceptually, a merge sort works as follows:

1. Divide the unsorted list into n sub-lists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sub-lists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

Example:



3.1.2 Quick Sort [Best: O(n log n), Worst: O(N^2)]

Quick sort is a divide and conquer algorithm which relies on a partition operation: to partition an array an element called a pivot is selected. It is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heap sort.

Quick sort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quick sort can then recursively sort the sub-arrays.

The steps are:

1. Pick an element, called a pivot, from the array.

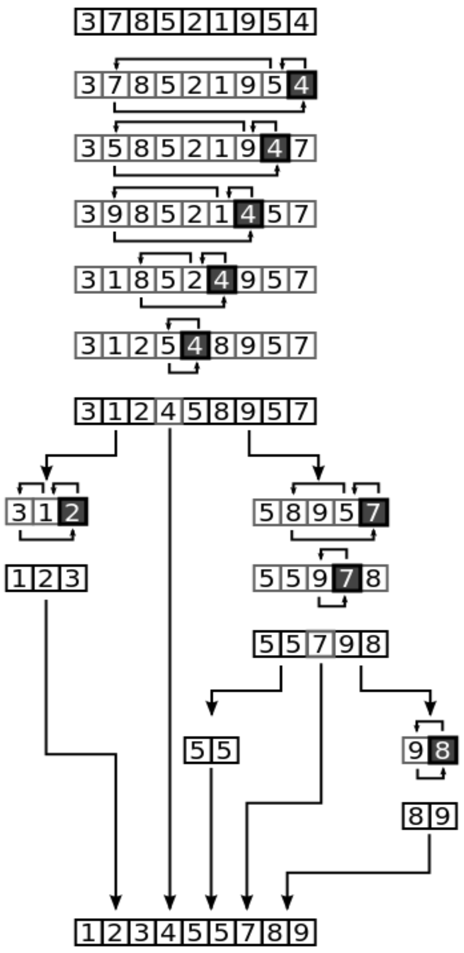
2. Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is arrays of size zero or one, which are in order by definition, so they never need to be sorted.

The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.

Example:

Following example explains the above steps:



3.2 Post Lab

Task 01

Implement Merge Sort and analyze its worst, best and average case complexity.

Task 02

Implement Quick Sort and analyze its worst, best and average case complexity.

3.3 References

4. **Introduction to Algorithms by *CLRS (3rd ed.)***
5. **Cplusplus.com**
6. **Class Notes**

Lab No.4 Searching Algorithms

OBJECTIVES OF THE LAB

In this lab, we will learn about some basic searching techniques and algorithms.

- *Linear Search*
- *Binary Search*

4.1 Pre Lab

4.1.1 Searching Algorithms

A searching algorithm is an algorithm which solves search problems i.e. to retrieve information stored within some data structure, or calculated in the search space of a problem domain. Examples of such structures include but are not limited to a linked list, an array data structure, or a search tree. The appropriate search algorithm often depends on the data structure being searched, and may also include prior knowledge about the data. Searching also encompasses algorithms that query the data structure, such as the SQL SELECT command.

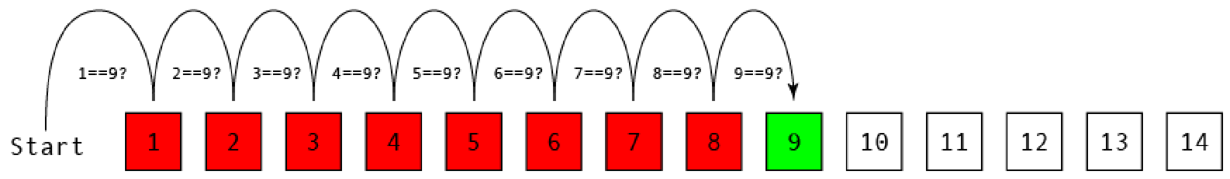
4.1.2 Linear Search [Best: O(1), Worst: O(N)]

Linear search or sequential search is a method for finding a target value within a list. It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched. Linear search runs in at worst linear time and makes at most n comparisons, where n is the length of the list. Linear search is rarely practical because other search algorithms and schemes, such as the binary search algorithm and hash tables, allow significantly faster searching for all but short lists.

Start from the leftmost element of arr[] and one by one compare x with each element of arr[]:

- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.

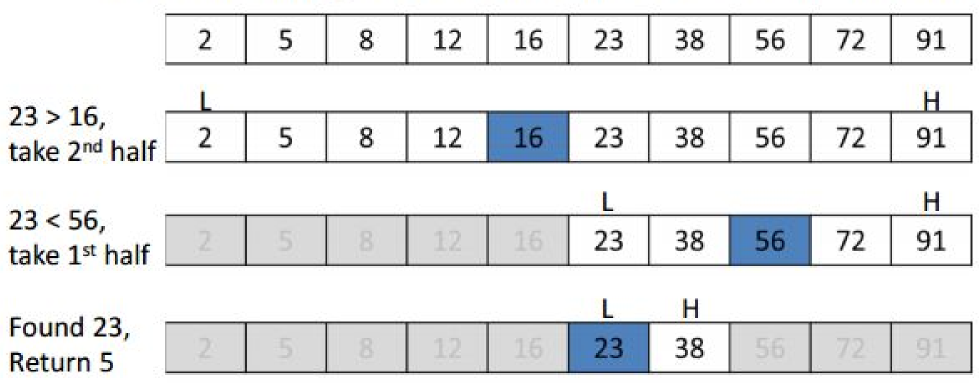
Example:



4.1.3 Binary Search [Best: O(1), Worst: O(log N)]

Binary search, also known as half-interval search, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array; if they are unequal, the half in which the target cannot lie is eliminated and the search continues on the remaining half until it is successful. If the search ends with the remaining half being empty, the target is not in the array. Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty. Binary search runs in at worst logarithmic time, making O(log n) comparisons, where n is the number of elements in the array, the O is Big O

notation, and log is the logarithm. Binary search takes constant (O(1)) space, meaning that the space taken by the algorithm is the same for any number of elements in the array.
Example:



4.2 Post Lab

Task 01

Implement Linear Search and analyze its worst, best and average case complexity.

Task 02

Implement Binary Search and analyze its worst, best and average case complexity.

4.3 References

7. **Introduction to Algorithms by *CLRS (3rd ed.)***
8. **Cplusplus.com**
9. **Class Notes**

Lab No.5 Probing Peak in Arrays

OBJECTIVES OF THE LAB

In this lab, we will learn about some basic techniques and algorithms to probe peak in.

- *One Dimensional Array*
- *Two Dimensional Array*

5.1 Pre Lab

5.1.1 Probing Peak Element in Array

A peak element is an element which is greater or equal to its neighbors. There can be multiple peak elements in an array.

5.1.2 Probing Peak Element in One Dimensional Array

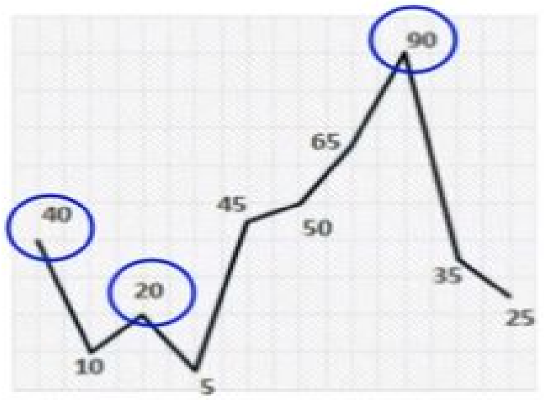
In one dimensional array the peak element has to be probed by comparison with only side elements (i.e. only row wise probing is required). Note: In sorted array the last element is always the peak.

Example:

Consider the following array:

40, 10, 20, 5, 45, 50, 65, 90, 35, 25

This array has peaks at 40, 20 and 90. Graphically:



5.1.2.1 Linear Probe in One Dimensional Array [Best: O(1), Worst: O(N)]

In linear algorithm we simply iterate over the array and find an element that is greater than or equal to all its neighbors and return it.

Example:

Consider the following array:

0, 10, 20, 5, 45, 50, 65, 90, 35, 25

i =1	***0 is not peak***
i = 2	***10 is not peak***
i = 3	***20 is peak return it***

5.1.2.2 Binary Probe in One Dimensional Array [Best/ Worst: n log (n)]

In binary algorithm we select a middle value x from the array and check for two possibilities:

1. X is peak element
2. X is not peak element

If X is peak then simply return X, if X is not peak then we have two possibilities:

1. Left neighbor is greater than X
2. Right neighbor is greater than X

If left neighbor is greater than X, we can always find a peak element on left of X. If right neighbor is greater than X, we can always find a peak element on right of X. Again divide the array in mid point and repeat the steps until peak element is found.

Example:
Consider the following array:

0, 10, 20, 5, 45, 50, 65, 90, 35, 25, 9

Step 1: Start with mid element i.e. 50, is 50 a peak element? No.
Check for both right and left neighbors of mid (50) and we see that 65 is greater than 50 and lies on the right.
Step 2: Take the right half as new array and take the new mid i.e. 35.
65, 90, 35, 25, 9
Step 3: Check the new mid element (35), is it a peak element? No.
Check for both right and left neighbors of mid (35) and we see that 90 is greater than 35 and lies on the left.
Step 4: Take the left half as new array and take the new mid i.e. 90.
65, 90, 35
Step 5: Check the new mid (90), is it a peak element? Yes. Return it.

5.1.3 Probing Peak Element in Two Dimensional Array

In two dimensional array the peak element has to be probed by comparison with not only side elements (i.e. row wise probing). But also the neighbor elements above and below are also required to be compared (i.e. column wie probing).

Example:
Consider the following array:

40, 10, 20, 54, 45, 50
50, 65, 90, 35, 25, 40
5, 45, 50, 90, 35, 25
35, 25, 40, 5, 45, 50

Those highlighted values are peak elements.

5.3 Post Lab

Task 01

Probe peak element in one dimensional array and analyze its worst, best and average case complexity.

Task 02

Probe peak element in two dimensional array and analyze its worst, best and average case complexity.

5.2 References

10. **Introduction to Algorithms by *CLRS (3rd ed.)***

11. **Cplusplus.com**

12. **Class Notes**

Lab No.6 Implementation of Lists using Arrays

OBJECTIVES OF THE LAB

In this lab, we will learn about the Data Structure, lists and their useful operations.

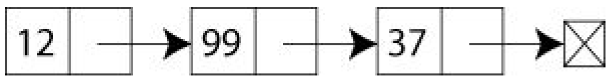
6.1 Pre Lab

6.1.1 Lists

A list or sequence is a data structure that represents a countable number of ordered values, where the same value may occur more than once.

Lists can be implemented using arrays and linked lists. In this lab we will do arrays implementation. When using arrays we have fixed size list where each element can be accessed directly using the index. As list is an abstract data type, we need to define operations. Some useful operations to be performed on the list are described in the next section.

Example:



As the name implies, lists can be used to store a list of elements. However, unlike in traditional arrays, lists can expand and shrink, and are stored dynamically in memory.

A finite set in the mathematical sense can be realized as a list with additional restrictions; that is, duplicate elements are disallowed and order is irrelevant. Sorting the list speeds up determining if a given item is already in the set, but in order to ensure the order, it requires more time to add new entry to the list. In efficient implementations, however, sets are implemented using self-balancing binary search trees or hash tables, rather than a list.

Lists also form the basis for other abstract data types including the queue, the stack, and their variations.

6.1.2 Operations on the List

- List createList (int size)
- Void Add (Object o, int index)

  Adding an object to list can be performed in 3 steps

  - Check for overflow
  - Check for bounds
  - Move all elements (from given index to index pointer) one index right
  - Add item to the given index
- Void Remove (int index)

  Removing an item from list can be performed in 4 steps

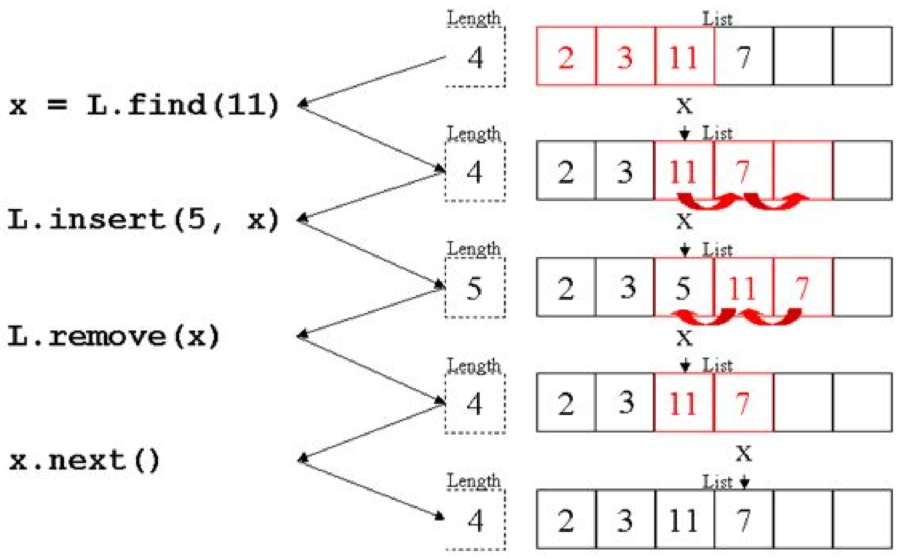
  - Check for underflow
  - Check for bounds
  - Remove item by shifting all the element (from given index to index pointer) one index left
  - Update index pointer to previous location
- Int Size ()

  Return index pointer.
- Bool IsEmpty ()

  If index pointer is -1 (list is empty) return true else return false.

- Object Get (int index)
  - Check for bounds
  - Return the object at the give index
- Int End ()
  -return the index pointer as it keeps track of last item in the list.
- Int Start ()
  - return 0 (starting index)

Example:



6.2 Post Lab

Task 01

Implement LIST using arrays. It should perform following operations

- Create (Creates LIST)
- Add (Adds item to the LIST at given index)
- Remove (Removes item from the List at provided index)
- waSize (Determines Size of the LIST)
- IsEmpty (Determines if LIST is empty or not)
- Get (Retrieves an Item from the LIST)
- End (Returns end of the LIST)
- Start (Return start of the LIST)

Task 02

Debug following code.

```
/* array implementation of LIST ADT */
#include <stdio.h>
#include <math.h>
#include <string.h>
#define MAX_LIST_SIZE 100
#define FALSE 0
#define TRUE 1
typedef struct {
        int number;
  char *string;
} ELEMENT_TYPE;
typedef struct {
        int last;
  ELEMENT_TYPE a[MAX_LIST_SIZE];
} LIST_TYPE;
typedef int WINDOW_TYPE;
/** position following last element in a list ***/
WINDOW_TYPE end(LIST_TYPE *list) {
  return(list->last+1);
}
/*** empty a list ***/
WINDOW_TYPE empty(LIST_TYPE *list) {
  list->last = -1;
  return(end(list));
}
/*** test to see if a list is empty ***/
int is_empty(LIST_TYPE *list) {
  if (list->last == -1)
  return(TRUE);
else
return(FALSE)
/*** position at first element in a list ***/
WINDOW_TYPE first(LIST_TYPE *list) {
  if (is_empty(list) == FALSE) {
  return(0);
else
     return(end(list));
}
/*** position at next element in a list ***/
WINDOW_TYPE next(WINDOW_TYPE w, LIST_TYPE *list) {
  if (w == last(list)) {
  return(end(list));
else if (w == end(list)) {
  error("can't find next after end of list"); }
else {
  return(w+1);
} }
/*** position at previous element in a list ***/
WINDOW_TYPE previous(WINDOW_TYPE w, LIST_TYPE *list) {
  if (w != first(list)) {
  return(w-1);
else {
   error("can't find previous before first element of
list");
return(w); } }
/*** position at last element in a list ***/
WINDOW_TYPE last(LIST_TYPE *list) {
return(list->last);}
```

6.3 References

13. **Introduction to Algorithms by** ***CLRS (3rd ed.)***

14. **Cplusplus.com**

15. **Class Notes**

Lab No.7 Implementation of Lists using Link List

OBJECTIVES OF THE LAB

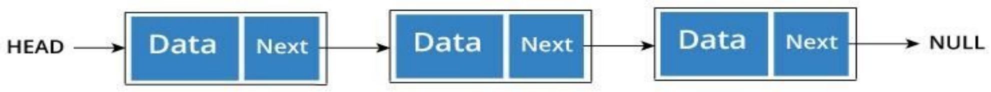
In this lab, we will learn about the data structure linked lists and their useful operations.

7.1 Pre Lab

7.1.1 Linked List

A linked list is similar in many ways to arrays. It is a series of connected "nodes" that contains the "address" of the next node. Each node can store a data point, which may be a number, a string or any other type of data.

7.1.2 Linked List Representations



You have to start somewhere, so we give the address of the first node a special name called *HEAD*. Also, the last node in the linked list can be identified because its next portion points to *NULL*.
How another node is referenced?
Some pointer magic is involved. Let's think about what each node contains:

- A data item
- An address of another node

We wrap both the data item and the next node reference in a struct as:

```
struct node
{
  int data;
  struct node *next;
};
```

Understanding the structure of a linked list node is the key to having a grasp on it.
Each struct node has a data item and a pointer to another struct node.

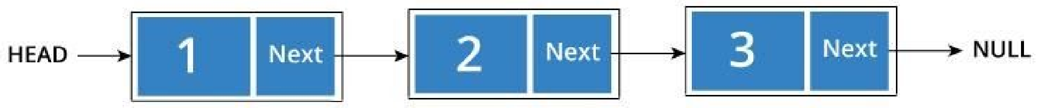
7.1.3 Operations on the Linked List

Let us create a simple Linked List with three items to understand how this works.

```
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data=3;
```

```
/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
```

If you didn't understand any of the lines above, all you need is a refresher on pointers and structs. In just a few steps, we have created a simple linkedlist with three nodes.



The power of linkedlist comes from the ability to break the chain and rejoin it. E.g. if you wanted to put an element 4 between 1 and 2, the steps would be:

- Create a new struct node and allocate memory to it.
- Add its data value as 4
- Point its next pointer to the struct node containing 2 as data value
- Change next pointer of "1" to the node we just created.

Doing something similar in an array would have required shifting the positions of all the subsequent elements.

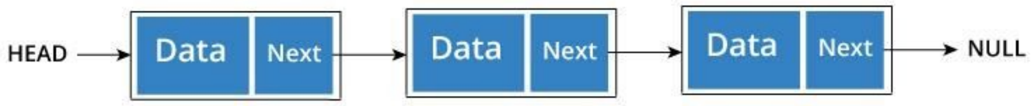
7.1.4 Types of Linked List

There are three common types of Linked List.

- Singly Linked List
- Doubly Linked List
- Circular Linked List

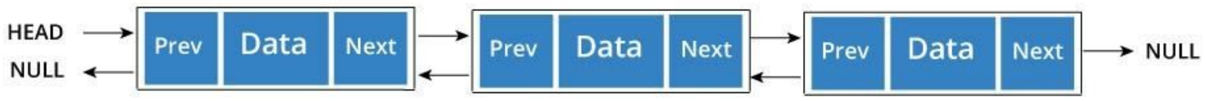
7.1.4.1 Singly Linked List

It is the most common. Each node has data and a pointer to the next node.



7.1.4.2 Doubly Linked List

We add a pointer to the previous node in a doubly linked list. Thus, we can go in either direction: forward or backward.

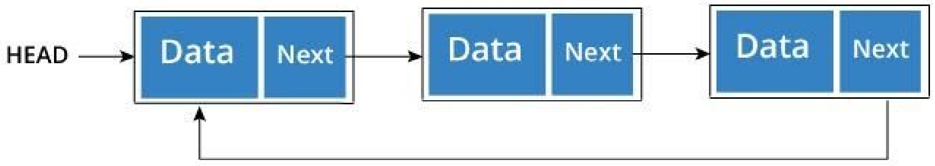


A node is represented as

```
struct node {
    int data;
    struct node *next;
    struct node *prev;
    }
```

7.1.4.3 Circular Linked List

A circular linked list is a variation of linked list in which the last element is linked to the first element. This forms a circular loop.



A circular linked list can be either singly linked or doubly linked.

- for singly linked list, next pointer of last item points to the first item
- In doubly linked list, prev pointer of first item points to last item as well.

7.1.5 Operations on linked list

- **Traverse a linked list**

  Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.
  When temp is *NULL*, we know that we have reached the end of linked list so we get out of the while loop.
  The output of this program will be:

  ```
  List elements are -     1 --->2 --->3 --->
  ```
- **Add elements to linked list**

  You can add elements to beginning, middle or end of linked list.

Add to beginning

  - Allocate memory for new node
  - Store data
  - Change next of new node to point to head
  - Change head to point to recently created node

Add to end

- Allocate memory for new node
- Store data
- Traverse to last node
- Change next of last node to recently created node

Add to middle

- Allocate memory and store data for new node
- Traverse to node just before the required position of new node
- Change next pointers to include new node in between

- **Delete from a linked list**
  You can delete either from beginning, end or from a particular position.

Delete from beginning

- Point head to the second node

Delete from end

- Traverse to second last element
- Change its next pointer to null

Delete from middle

- Traverse to element before the element to be deleted
- Change next pointers to exclude the node from the chain

7.2 Post Lab

Task 01

Implement singly linked list with following operations:

a) Traverse
b) Delete
c) Print
d) Add

Task 02

Implement a single linked list with following operations

a) Insert at start of the list (preappend data)
b) Insertion at end (append data)

c) Insertion at nth location
d) Deletion from start of the list
e) Deletion the end of the list
f) Insertion from nth location
g) Search for a key
h) Update list
i) Empty list
j) Calculate size of the list

Task 03

Implement doubly linked list and perform operations mentioned on it from Task 02

Task 04

Implement circular linked list and perform operations mentioned on it from Task 02

7.3 References

16. **Introduction to Algorithms by *CLRS (3rd ed.)* (Chapter 09)**
17. **Class Notes**

Lab No.8 Stacks and Queues

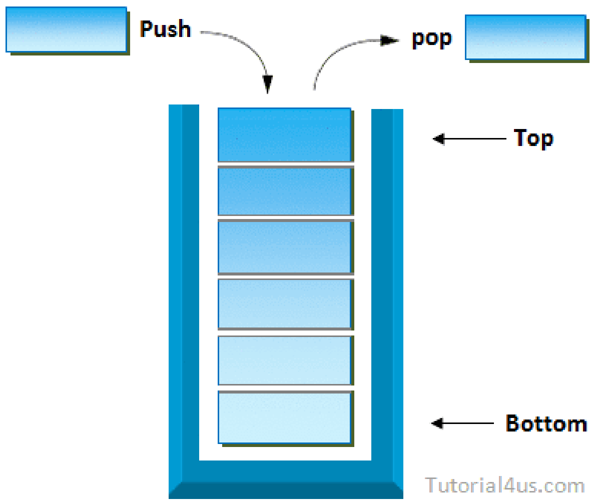
OBJECTIVES OF THE LAB

In this lab, we will implement and perform basic operations on Stacks and Queues data structures.

8.1 Pre Lab

8.1.1 Stack

A stack is a useful data structure in programming. It is just like a pile of plates kept on top of each other.



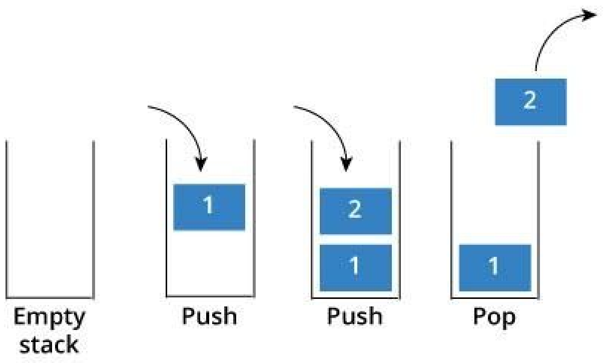
Think about the things you can do with such a pile of plates

- Put a new plate on top
- Remove the top plate

If you want the plate at the bottom, you have to first remove all the plates on top. Such kind of arrangement is called **Last In First Out** - the last item that was placed is the first item to go out.

8.1.2 Stack in Programming Terms

In programming terms, putting an item on top of the stack is called "push" and removing an item is called "pop".



In the above image, although item 2 was kept last, it was removed first - so it follows the Last In First Out(LIFO) principle.

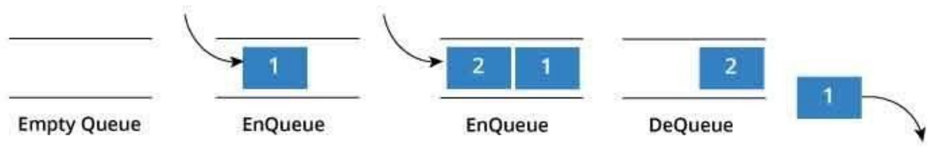
8.1.3 Operations on Stack

A stack is an object or more specifically an abstract data structure(ADT) that allows the following operations:

- **Push:** Add element to top of stack
- **Pop:** Remove element from top of stack
- **IsEmpty:** Check if stack is empty
- **IsFull:** Check if stack is full
- **Peek:** Get the value of the top element without removing it

8.1.4 Queue

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.
Queue follows the **First In First Out(FIFO)** rule - the item that goes in first is the item that comes out first too.



In the above image, since 1 was kept in the queue before 2, it was the first to be removed from the queue as well. It follows the FIFO rule.
In programming terms, putting an item in the queue is called an "enqueue" and removing an item from the queue is called "dequeue".

8.1.5 Operations on Queue

A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations:

- **Enqueue:** Add element to end of queue
- **Dequeue:** Remove element from front of queue
- **IsEmpty:** Check if queue is empty
- **IsFull:** Check if queue is full
- **Peek:** Get the value of the front of queue without removing it

8.2 Post Lab

Task 01

Implement stacks with following operations using arrays using Arrays

a) Creation
b) Pop
c) Push
d) Top
e) Size
f) Empty

Task 02

Implement stacks with following operations using linked Lists

a) Creation
b) Pop
c) Push
d) Top
e) Size
f) Empty

Task 03

Implement queues with following operations using arrays

a) Creation
b) Enqueue
c) Dequeue
d) Front
e) Rear
f) Size
g) Empty

Task 04

Implement queueswith following operations using linked list

a) Creation
b) Enqueue
c) Dequeue
d) Front
e) Rear
f) Size
g) Empty

7.4 References

18. **Introduction to Algorithms by *CLRS (3rd ed.)* (Chapter 10)**

19. **Class Notes**

Lab No.9 Circular Queues

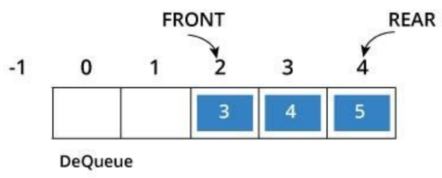
OBJECTIVES OF THE LAB

In this lab, we will implement and perform basic operations on Circular Queues data structures.

9.1 Pre Lab

9.1.1 Circular Queues

Circular queue avoids the wastage of space in a regular queue implementation using arrays.

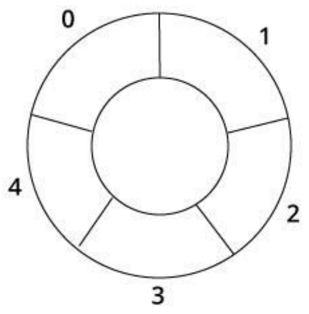


As you can see in the above image, after a bit of enqueueing and dequeueing, the size of the queue has been reduced. The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued.

9.1.2 How Circular Queue Works

Circular Queue works by the process of circular increment i.e. when we try to increment any variable and we reach the end of queue, we start from the beginning of queue by modulo division with the queue size.

i.e. if REAR + 1 == 5 (overflow!),
REAR = (REAR + 1)%5 = 0 (start of queue)



9.1.3 Operations on Circular Queues

In programming terms, putting an item on top of the stack is called **"push"** and removing an item is called **"pop".** Two pointers called ***FRONT*** and ***REAR*** are used to keep track of the first and last elements in the queue.

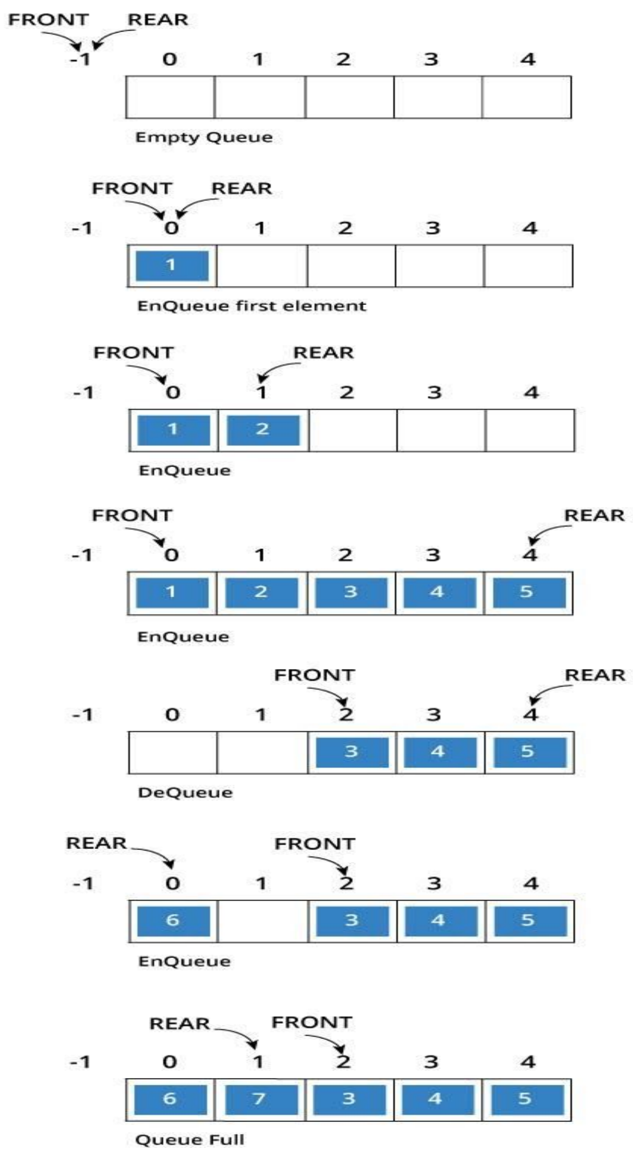
- When initializing the queue, we set the value of **FRONT** and **REAR** to **-1.**
- On **enqueing** an element, we circularly increase the value of **REAR** index and place the new element in the position pointed to by **REAR.**
- On **dequeueing** an element, we return the value pointed to by **FRONT** and circularly increase the **FRONT** index.

- Before **enqueing,** we check if queue is already full.
- Before **dequeuing**, we check if queue is already empty.
- When **enqueing** the first element, we set the value of **FRONT** to **0**.
- When **dequeing** the last element, we reset the values of **FRONT** and **REAR** to **-1.**

However, the check for full queue has a new additional case:

- **Case 1:** *FRONT* = 0 && REAR == SIZE - 1
- **Case 2:** FRONT = REAR + 1

The second case happens when *REAR* starts from 0 due to circular increment and when its value is just 1 less than *FRONT*, the queue is full.



9.2 Post Lab

Task 01

Implement following operations of circular queue using arrays

a) Creation
b) Enqueue
c) Dequeue
d) Front
e) Rear
f) Size
g) empty

Task 02

Implement circular queue with operations from Task1

9.3 References

20. **Introduction to Algorithms by *CLRS (3rd ed.)* (Chapter 10)**

21. **Class Notes**

Lab No.10 Binary Search Tree (BST)

OBJECTIVES OF THE LAB

In this lab, we will implement and perform basic operations on Binary Search Tree Data Structures.

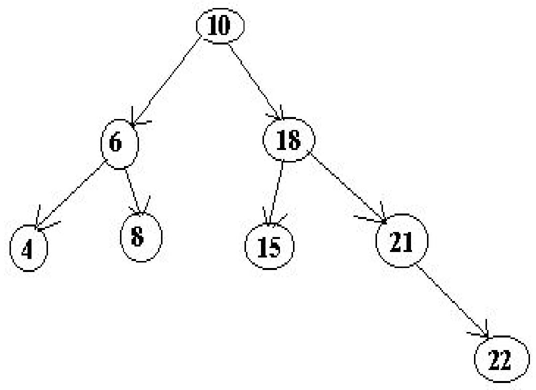
10.1Pre Lab

10.1.1 Binary Search Tree

A BST is a binary tree of nodes ordered in the following way:

1. Each node contains one key (also unique)
2. The keys in the left subtree are < (less) than the key in its parent node
3. The keys in the right subtree > (greater) than the key in its parent node
4. Duplicate node keys are not allowed.

   Here is an example of a BST



10.1.2 Operations on Binary Search

There are a number of operations on BSTs that are important to understand. We will discuss some of the basic operations such as how to insert a node into a BST, how to delete a node from a BST and how to search for a node in a BST.

10.1.2.1 Inserting a node

A naive algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root. We continue this process (each node is a root for some sub tree until we find a null pointer (or leaf node) where we cannot go any further. We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node. We note that a new node is always inserted as a leaf node. A recursive algorithm for inserting a node into a BST is as follows. Assume we insert a node N to tree T. If the tree is empty, then we return new node N as the tree. Otherwise, the problem of inserting is reduced to inserting the node N to left of right sub trees of T, depending on N is less or greater than T. A definition is as follows:

```
Insert(N, T)  = N   if T is empty
              = insert(N, T.left)  if  N < T
              = insert(N, T.right) if  N > T
```

10.1.2.2 Searching for a node

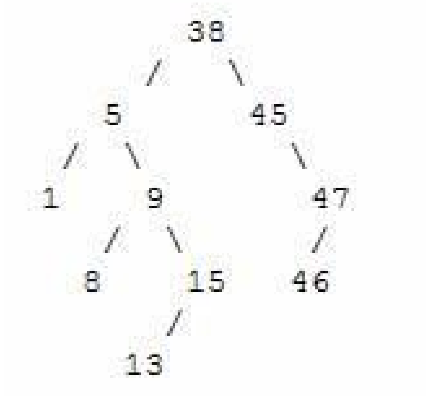
Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node). A recursive definition of search is as follows. If the node is equal to

root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for T.left or T.right, depending on $N < T$ or $N > T$. Search should return a true or false, depending on the node is found or not.
A recursive definition is as follows:

Search(N, T) = false if T is empty
= true if T = N
= search(N, T.left) if N < T
= search(N, T.right) if N > T

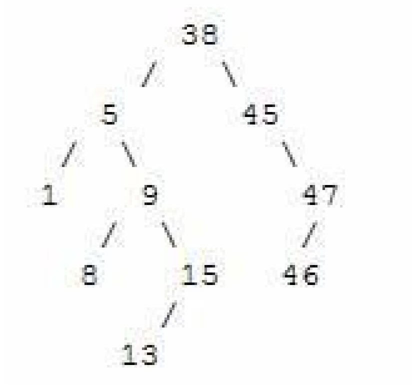
10.1.2.3 Deleting a node

A BST is a connected structure. That is, all nodes in a tree are connected to some other node. For example, each node has a parent, unless node is the root. Therefore deleting a node could affect all sub trees of that node. For example, deleting node 5 from the tree could result in losing sub trees that are rooted at 1 and 9. Hence we need to be careful about deleting nodes from a tree. The best way to deal with deletion seems to be considering special cases. What if the node to delete is a leaf node? What if the node is a node with just one child? What if the node is an internal node (with two children)? The latter case is the hardest to resolve. But we will find a way to handle this situation as well.



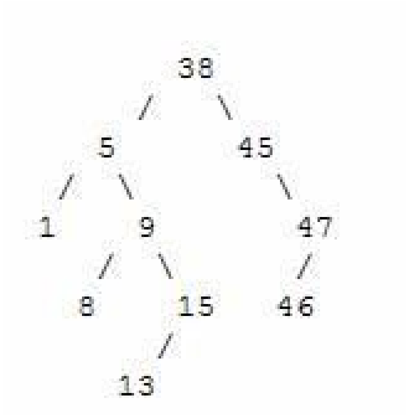
Case 1: The node to delete is a leaf node

This is a very easy case. Just delete the node and we are done. For example deleting leaf node: 13



Case 2: The node to delete is a node with one child.

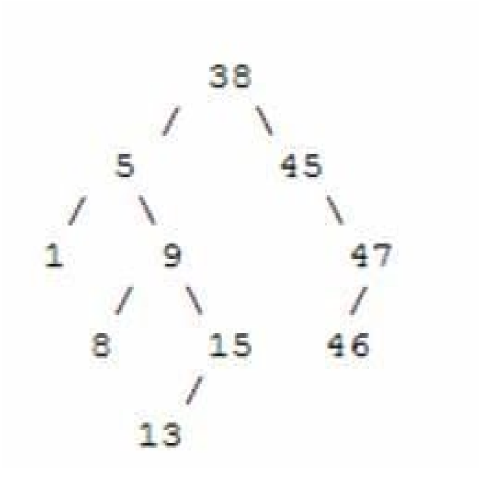
This is also not too bad. If the node to be deleted is a left child of the parent, then we connect the left pointer of the parent (of the deleted node) to the single child. Otherwise if the node to be deleted is a right child of the parent, then we connect the right pointer of the parent (of the deleted node) to single child. For example deleting node 15 and connecting the right pointer to node 13.



Case 3: The node to delete is a node with two children

This is a difficult case as we need to deal with two sub trees. But we find an easy way to handle it. First we find a replacement node (from leaf node or nodes with one child) for the node to be deleted. We need to do this while maintaining the BST order property. Then we swap leaf node or node with one child with the node to be deleted (swap the data) and delete the leaf node or node with one child (case 1 or case 2)

Next problem is finding a replacement leaf node for the node to be deleted. We can easily find this as follows. If the node to be deleted is N, then find the largest node in the left sub tree of N or the smallest node in the right sub tree of N. These are two candidates that can replace the node to be deleted without losing the order property. For example, consider the following tree and suppose we need to delete the root 38.



Then we find the largest node in the left sub tree (15) or smallest node in the right sub tree (45) and replace the root with that node and then delete that node.

10.2 Post Lab

Task 01

Implement binary search tree with following operations

a) Creation
b) Insertion
c) Find minimum
d) Find Maximum
e) Delete Element
f) Post order traversal
g) Pre order traversal
h) In order traversal
i) Height of tree

10.3 References

22. **Introduction to Algorithms by *CLRS (3rd ed.)* (Chapter 10)**

23. **Class Notes**

–

Lab No.11 Hash Tables

11.1 Objectives of the lab

Implementing and performing basic operations on Hash Table data structure.

11.2 Pre Lab

11.2.1 Hash Table

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.
Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash

technique to generate an index where an element is to be inserted or is to be located from.

11.2.2 Hash Function

In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.

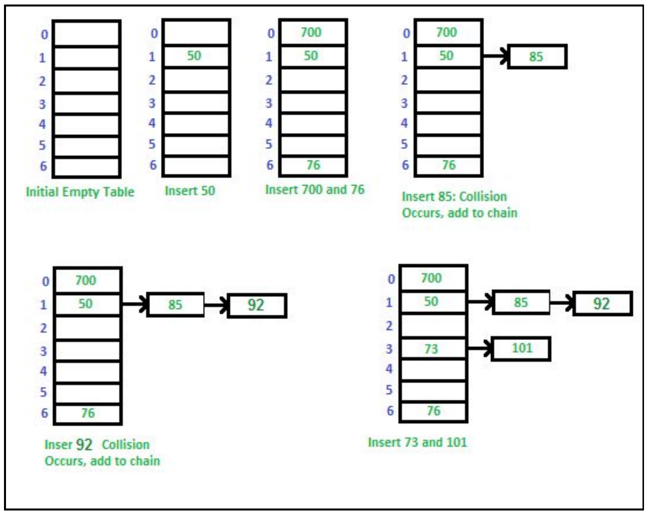
A good hash function should have following properties

1) Efficiently computable.
2) Should uniformly distribute the keys (Each table position equally likely for each key)

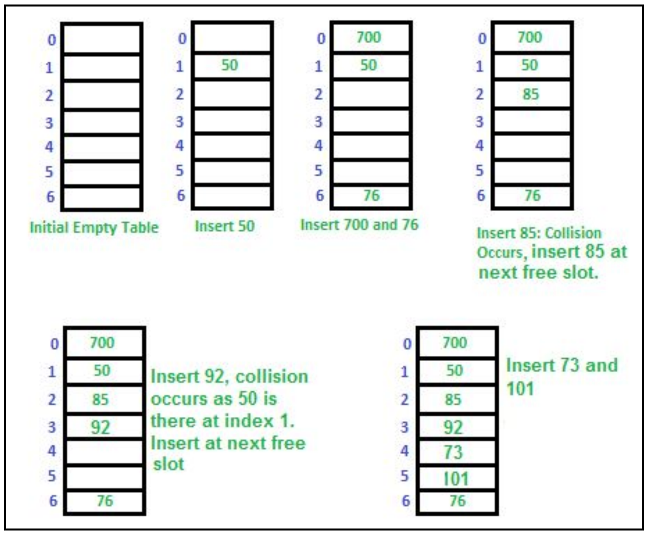
For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

11.2.3 Collision Handling

- **Chaining:** 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.


Initial Empty Table
Insert 50
Insert 700 and 76
Insert 85: Collision Occurs, add to chain
Inser 92 Collision Occurs, add to chain
Insert 73 and 101

- **Open Addressing:** 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.


Initial Empty Table
Insert 50
Insert 700 and 76
Insert 85: Collision Occurs, insert 85 at next free slot.
Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot
Insert 73 and 101

11.3 Post-Lab

11.3.1 Activity

Implement hash table with following operations

a) Hashing
b) Insertion
c) Deletion
d) Searching
e) Rehashing

Note: Generally we do re-hashing when hashtable is filled beyond certain number-Load Factor. While doing re-hash, we increase the size of hashtable and move the contents to new hash table.

11.4 References:

1 **Class notes**

2 **Introduction to algorithms by CLRS (Chapter 11)**

Lab No.12 Graph: Adjacency Matrix

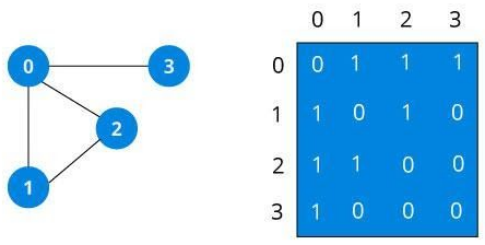
12.1 Objectives of the lab

Implementing graph using adjacency matrix.

12.2 Pre Lab

12.2.1 Adjacency Matrix
An adjacency matrix is a way of representing a graph G = {V, E} as a matrix of booleans.

12.2.2 Adjacency Matrix Representation
The size of the matrix is `VxV` where `V` is the number of vertices in the graph and the value of an entry `Aij` is either 1 or 0 depending on whether there is an edge from vertex i to vertex j.
The image below shows a graph and its equivalent adjacency matrix.



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 \end{array}$$

In case of undirected graph, the matrix is symmetric about the diagonal because of every edge `(i,j)`, there is also an edge `(j,i)`.

12.2.3 Adjacency Matrix Operations (undirected graph)

- **void addEdge(int i, int j)**
  set Aij to 1 (indicates that edge exists)
- **void removeEdge (int i, int j)**
  set a[i][j] to 0 (indicates edge does not exist)
- **bool isEdge (int I, int j)**
  return Aij

12.3Post-Lab

12.3.1 Activity
Implement graphs with the following operations

a) Graph Creation
b) Adding Vertex to Graph
c) Removal of Vertex from the graph
d) Checking whether an edge exist between two vertices
e) Printing graph

12.4 References:

1 **Class notes**