selection sort code searches an array looking for the smallest element in the array. Then, the

smallest element is swapped with the first element of the array. The process is repeated for the

sub-array beginning with the second element of the array. Each pass of the array results in one

element being placed in its proper location. When the sub-array being processed contains one

element, the array is sorted. Write C++ code for this selection sort and output must be stored in

“output.txt” file (having all passes

selection sort code searches an array looking for the smallest element in the array. Then, the

smallest element is swapped with the first element of the array. The process is repeated for the

sub-array beginning with the second element of the array. Each pass of the array results in one

element being placed in its proper location. When the sub-array being processed contains one

element, the array is sorted. Write C++ code for this selection sort and output must be stored in

“output.txt” file (having all passes

# Department of Computing

**CS250: Data Structure and Algorithms**

**Class: BSCS 11A&C**

**Course Instructor: Dr. Yasir Faheem**

# Lab 2: Pointers & Array-based Implementation of Lists

**Part One: Revision of Pointers in C++**

**Introduction**

This lab is about pointers, memory occupied by variables, and dynamic vs static memory allocation.

**Objectives**

This lab will revise the concepts learnt by students in the previous semesters.

**Tools/Software Requirement**

Visual Studio C++, Eclipse C++ or any other IDE.

**Description**

Pointers are used to point towards a particular memory address. In this lab, we will use the pointers and perform tasks with the help of them.

**Lab Tasks**

You are required to upload the lab tasks on the LMS and the name of that tasks must be in this format YourFullName\_reg#.cpp

Remember to comment your code properly. Inappropriate or no comment will result in the deduction of marks.

**Task 1:** Write output of the following C++ codes in your document without executing it.

**Example code a)**

int a;

int b;

int \*p=&a;

int \*q=&b;

a=20;

b=35;

p=q;

\*p=83;

cout<<"a : "<<a<<" b: "<<b<<endl;

cout<<\*p<<" "<<\*q<<endl;

**Output:**

**a : 20 b: 83**

**83 83**

**Example code b)**

int x[4] = {0,4,6,9};  
int \*p, a=3;  
p=x;  
(\*p)++;  
cout<<\*p<<endl;  
cout<<\*(p+1)<<endl;  
p++;  
\*p=\*p+a;  
cout<<\*p<<endl;  
p=p+2; //What is happening here?  
cout<<\*p<<endl;

**Output:**

1

4

7

9

**Example code c)**

int a, \*p, \*q;

int arr[4]= {0};

p=arr;

q=p;

\*p=4;

for(int i=0; i<3; i++){

a=\*p;

p++;

\*p=(a+i);

}

for (int j=0; j<4; j++){

cout<<\*q<<" ";

q++;

}

**Output:**

4 4 5 7

**Task 2:**

int a=5, b=10;

int \*pa=&a; //pa and pb are pointer variables of type int.

int \*pb=&b;

int \*\*ppa=&pa; //ppa and ppb are called double pointers or pointers-to-pointers.

int \*\*ppb=&pb;

1. Write code of a function that swaps values of variables a and b. Input to the function should be the address of both the variables.
2. Write code of a function that swaps values of pointer variables pa and pb. Input to the function should be the address of both the pointer variables.
3. Write code of a function that swaps values of the variables a and b using pointer-to-pointer variables ppa and ppb.

**Task 3:**

int list[5]={3,6,9,12,15};

int \*pArr= list;

Your task is to write a piece of code that prints all values stored in the array **list** using only pointer variable pArr. Do not use the conventional way of printing values by numbering indexes.

**Task 4:** Write code to find the memory in bytes occupied by int, long, double, float and char.

**Recap: Static vs Dynamic Arrays**

Consider the two variants of declaring arrays below. Memory for the first variant gets allocated on the activation stack. The lifetime of an array created using method-A depends on its scope. If it is defined globally, its life is equal to the lifetime of the application. If it is declared in a function, memory for it gets allocated on the stack when the function gets called. It gets deallocated when the function call terminates. All the data related to the function call including the array gets removed from the stack. On the other hand, memory for the array created using new operator gets allocated on the heap at runtime. The lifetime of such an array is at max equal to the execution time of the application. If the array is not required during the execution of the program, the memory allocated to it can be freed using **delete []** command.

**Method A:**

const int size=5;

int x[size];

for (int i = 0; i < size; i++)

{

//cout << "x[" << i << "] = ";

x[i] = i + 1;

}

**Method B**

int size; // Note that size variable is const in variant A whereas it isn’t in variant B. Find out the logic behind it.

cout << "Enter size of array: ";

cin >> size;

int \*x = new int[size];

for (int i = 0; i < size; i++)

{

//cout << "x[" << i << "] = ";

x[i] = i + 1;

}

**Part Two: Array-based Implementation of List Data Structure**

**Introduction**

We have studied List as Abstract Data Type in the class. We have discussed various operations that can be performed on a list. The objective of this lab is to implement an array-based list. First, you shall implement static array-based list which is not resizable. Then, you shall implement a dynamic version of an array-list; you shall allocate memory for array at runtime using the **new** operator.

Your task is to implement all the following operations.

1. bool IsEmpty() operation. It returns true value if the list is empty. Otherwise, false.
2. bool isFull() function. It returns true value if the list is full. False otherwise.
3. InsertSorted(value) a value at its logical position in a list.
4. CreateRoom(int pos); If a new value has to be inserted into a list at position pos, this function creates room for it, by shifting all values from position pos till last one index to the right.
5. FillGap(int pos); If an existing value has to be deleted from position pos, this function fills the gap by shifting all values from pos till last element to the left side.
6. DynamicExpansion( ); If an array-list is already full, this function dynamically creates a new array of size greater than the current size of the array, copies all contents from the older array to the new one, updates the pointer which stores the address of an array, and lastly deletes the older array using **delete** ptr[] command.
7. DynamicReduction(); If the size of a list falls below 50% capacity of the array after a deletion operation, this function dynamically creates a new array of size smaller than the current size of the existing array, copies all contents from the older array to the new one, updates the pointer that stores the address of an array, and lastly deletes the older bigger array using **delete** ptr[] command.

**Task 1: Implement a version of an array-list in which the array is static.**

struct staticArrayList{

int array[size];

bool isEmpty();

bool isFull();

void insertAtposition(int value, int position);

void insertSorted(int value);

void printList();

void DeleteValue(int value);

void DeleteValueAtPosition(int pos); // deletes a value if a user enters a valid position (1 <= pos <= size of a list.)

**Note:** The size of a list and that of an array are different things. Moreover, an element at position pos in a list is stored at index pos-1 in an array.

}

**Task 2: Implement a version of an array-list in which the array is dynamic**. Memory for an array of capacity *size*can be allocated at runtime using new operator. It can be deallocated using **delete** operator. Moreover, array-list can be resized if it becomes full, or the number of elements in it fall below a certain threshold. In addition, in this version of the program you shall assume that we are maintaining a list sorted in the ascending order; InsertAtEnd() and InsertAtPosition() functions are no longer required in this version of the program. You may implement a new function named InsertSorted(newValue). It should find the logical position of the newValue in the list, create room for it, if required, by calling the makeRoom(pos) function, and then insert it.

struct DynamicArrayList{

int \*pArray[];

const int size;

pArrayList = new int [size];

int array[size];

bool isEmpty();

bool **isArrayFull();**

void insertAtposition(int value, int position);

void insertSorted(int value);

void printList();

void DeleteValue(int value);

void DeleteValueAtPosition(int pos); // deletes a value if a user enters a valid position (1 <= pos <= size of a list)

void DynamicExpansion(); //Increases size of the array when the list becomes full.

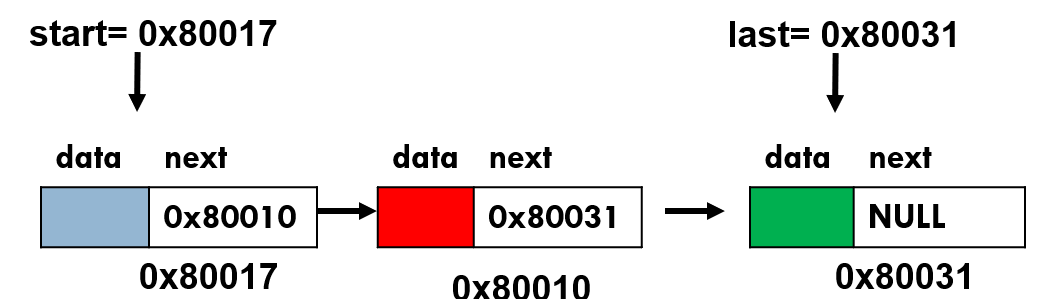
void DynamicReduction(); //decreases size of the array when the size of a list becomes less than 50% after deletion operation

}

# Singly Linked Lists

The objective of this lab session is to acquire skills in working with singly linked lists. The task is to first implement the following operations:

1. Create an empty linked list; // do so in the constructor.
2. bool IsEmpty(); // checks whether the list is empty or not. Returns true if empty and false otherwise.
3. InsertAtFront(value); // takes input from a user and inserts it at the front of a list
4. InsertAtEnd(value) ; // takes input from a user and inserts it at the tail end of a list
5. PrintList();
6. InsertSorted(value); //To maintain a sorted list, you shall implement this function. Note that if you are maintaining a sorted list then do not call InsertAtFront(value) and InsertAt(Front) functions in the main function.
7. Search(value); This function shall search value in a list. If found, we will need to store two addresses:
   1. Address of the node in which the searched value is found in a pointer variable named Loc\_; we will store NULL in Loc\_ in case value is not found.
   2. Address of the node which is logical predecessor of value in a list.
      1. The Search() provides functionality for other operations such as insertion in a sorted list, deleting a value, modifying a value, printing it etc.
8. Delete(value); // searches value and then deletes it if found.
9. DestroyList(); // Distroys all nodes of the list leaving the list in empty state.



**Declare Node Class:** The data structure that will hold the elements of the list is called **Node.** Declare it as follows:

**class** ListNode{

**public**:

**int** data;

ListNode \*next;

};

**Declare class Linked List:** Now, declare your main class LinkedList. Inside this class, you shall define all key functions to implement all operations of a linked list.

**class** LinkedList{

**public**:

ListNode \*start; // special variable which stores address of the head node.

ListNode \*last; // special variable which stores address of the last node.

ListNode \*PredLoc\_; //to be used by Search(value) method to store address of logical predecessor of value in a list.

ListNode \*Loc\_; //to be used by Search(value) method to store address of the node containing the searched value in a list. If it is not found it contains NULL.

}

1. **Creating a LinkedList**

In order to create an empty list, assign NULL value to start pointer variable.

LinkedList(){

start=NULL;

PredLoc\_=NULL;

Loc\_=NULL;

}

1. **Bool IsEmpty() function**

By checking content of the special pointer variable start/first, this function should return true value if the list is empty and false otherwise.

1. **Inserting a value at the Front of a list**

First, Reserve space for a new node to be inserted in the list by creating object of class ListNode and storing its address in a temporary pointer variable.

ListNode \*newnode = **new** ListNode();

Now store value in data part of the new node:

*newnode->data=value;*

If you are inserting in an empty list, you should update both the front and last pointer variables as follows:

start=newnode;

last=newnode;

Finally, link newnode at front of the linked list via following two statements:

newnode->next=start;

start=newnode;

1. **Inserting a value at the tail end of a list**

First, Reserve space for a new node to be inserted in the list by creating object of class ListNode and storing its address in a temporary pointer variable.

ListNode \*newnode = **new** ListNode();

Now store value in data part of the new node:

*newnode->data=value;*

If you are inserting in an empty list, you should update both the front and last pointer variables as follows:

start=newnode;

last=newnode;

Finally, link newnode at front of the linked list via following two statements:

last->next=newnode;

last=newnode;

1. **void PrintList()**

This function prints all elements of a linked list starting from the first one.

ListNode \*temp = start;

If(list not empty){

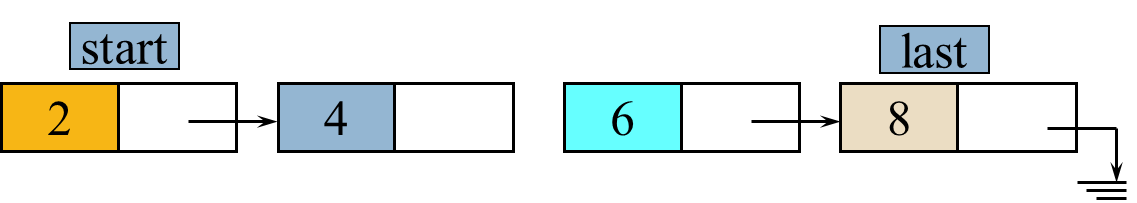
While(not end of list){

cout<<temp->data<<” “;

// advance temp to successor node.

}

}



1. **Void Search( value)**

As discussed in the class, we will implement a general-purpose search function which will provide functionality to other operations like insertion, deletion, modification etc. This function shall take a value as argument from the user and then search it in the list. You should use two node pointer variables namely ploc and loc in this function.

* 1. In the variable loc, save address of the node in which the searched value is found. In case, the searched value does not exist, save NULL value in loc.
  2. In the variable ploc, we shall store address of the logical predecessor node of the searched value.

**Void search(value){**

Initialize loc & ploc

Loc= address of head node

Ploc = address of logical predecessor of head node. Note that first node has no predecessor. So, we will always initialize ploc to NULL.

For the moment assume that we are maintaining a list sorted in the ascending order. Search value until either 1) we reach the end of the list or 2) logical position of the value is passed.

while (loc!=NULL and loc->data < value){

Advance both ploc and loc

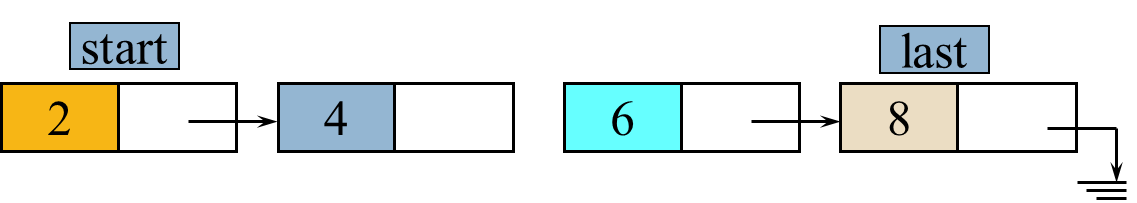
}

If(loc!NULL & loc->data!=value)

Loc=NULL; //as value is not found so set loc equal to null.

} //end of search function.

After execution of the search(value) method, there are four possible combinations of loc and ploc



|  |  |  |  |
| --- | --- | --- | --- |
| **Search example** | **Ploc** | **Loc** | **Interpretation** |
| 1 | Null | Null | Searched value not found and its logical position is at the front of the list |
| 2 | Null | Non-null | Value found in the head node of the list |
| 4/6 | Non-null | Non-null | Loc=non-null implies the searched value has been found. Ploc=non-null implies the value is not in the head node; it might be in any node other than the head node.  **Special case:** loc ==last implies value found in the last node. |
| 10 | Non-null | Null | Loc=null implies searched value not found. Ploc=non-null implies its logical position is not at the front. |

1. **Insertion in a Sorted List**

For the moment, assume duplications are not allowed in the list. You have to insert new value after call to search function by considering the above mentioned four possible combinations of loc and ploc pointer variables.

InsertSorted(value){

Search(value)

If (value already exists) //check using loc

Return without insertion and print a message

else{

If(position of value is as head node)

Insert value at front.

Else //Insert newnode after ploc.

If insertion at the tail end, do update last.

Else insert value after ploc in the middle.

}

1. **Delete a Value**

Find value using search method and if a node containing the searched value is found, then delete it from the linked list. Also free the allocated memory.

Delete(value){

//if list empty then return

Search(value)

If(value is found){ //check loc

If(value is in the head node)//check ploc {

//delete head node and free memory

}

else{

//update link using ploc

ploc->next = loc->next;

//finally free memory using delete command.

delete loc;

}

}

1. **Destroy a Linked List**

This method should destroy all nodes of a linked list making it empty. It should also free space allocated for all the nodes.

**Hint:** Save address of current head node in a temporary pointer variable. Advance start variable to the second node so that it becomes new head node. Then, delete current head node using the temporary pointer variable.