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 9AB**

# 

# Lab 1: 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:**

****

**Example code b)**

int x[4] = {0,4,6,9};

int \*p, a=3;

p=x;

(\*p)++;

cout<<\*p;

cout<<\*(p+1)<<endl;

cout<<\*p=\*p+a<<endl;

cout<<\*p<<endl;

p++;

cout<<\*p<<endl;

**Output:**

**Example code c)**

int a, \*p, \*q;

int arr[4]= {0};

p=arr;

q=p;

\*p=4;

for(int i=0; i<2; i++){

x=\*p;

p++;

\*p=(a+i);

}

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

cout<<\*q<<" ";

q++;

}

**Output:**

**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.

**Code:**

#include <iostream>

using namespace std;

int swap(int \*p1, int \*p2);

int main()

{

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;

cout << " Before using swap function" << endl;

cout << a << endl;

cout << b << endl;

swap(pa, pb);

cout << " After swap function" << endl;

cout << a << endl;

cout << b << endl;

}

int swap (int \*p1, int \*p2)

{

int temp;

temp = \*p1;

\*p1 = \*p2;

\*p2 = temp;

return 0;

}**Output**

****

1. 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.

**Code**

# include <iostream>

using namespace std;

int swap(int\*\*p1, int \*\*p2);

int main()

{

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;

cout << " Before using swap function" << endl;

cout << \*pa << endl;

cout << \*pb << endl;

swap(ppa, ppb);

cout << " After swap function" << endl;

cout << \*pa << endl;

cout << \*pb << endl;

}

int swap(int \*\*p1, int \*\*p2)

{

int temp;

temp = \*\*p1;

\*\*p1 = \*\*p2;

\*\*p2 = temp;

return 0;

}

**Output**

****

1. Write code of a function that swaps values of the variables a and b using pointer-to-pointer variables ppa and ppb.

# include <iostream>

using namespace std;

int swap(int\*\*p1, int \*\*p2);

int main()

{

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;

cout << " Before using swap function" << endl;

cout << a << endl;

cout << b << endl;

swap(ppa, ppb);

cout << " After swap function" << endl;

cout << a << endl;

cout << b << endl;

}

int swap(int \*\*p1, int \*\*p2)

{

int temp;

temp = \*\*p1;

\*\*p1 = \*\*p2;

\*\*p2 = temp;

return 0;

}

**Output**

**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.

**Code**

# include <iostream>

using namespace std;

int main()

{

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

int \*pArr = list;

for (int i = 0; i < 5; i++)

{

cout << \*pArr << endl;

\*(pArr++);

}

}

**Output**

****

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

# include <iostream>

using namespace std;

int main()

{

cout << sizeof(int) << endl;

cout << sizeof(double) << endl;

cout << sizeof(float) << endl;

cout << sizeof(char) << endl;

}



**Part Two: Static vs Dynamic Arrays**

Consider two variants of declaring arrays below. Memory for the first variant gets allocated on the Stack. The lifetime of an array created using the 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 no more required, the memory allocated for it can be freed using **delete []** command.

**Method A:**

const int size;

cout << "Enter size of array: ";

cin >> size; //In c++, cin command is used to take input from users.

int x[size];

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

{

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

x[i] = i + 1;

}

**Output**

x[0] = x[1] = x[2] = x[3] = x[4] = Press any key to continue . . .

**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;

}

Enter size of array: 4

x[0] = x[1] = x[2] = x[3] = Press any key to continue

**Part Three: 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);

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 an array are different things. Moreover, an element a location pos in a list is stored at index pos-1 in the array.

}

#include <iostream>

using namespace std;

const int size = 10;

struct staticArrayList{

int array[size]= { 8, 5, 18, 4, 7, 6, 24, 3, 2 };

//print\_list

void print\_list(int array[size\_array])

{

int length=0;

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

{

length = array[i];

if (length != NULL)

{

cout << array[i]<<endl;

}

}

}



//prints only the content of list

**//is empty**

bool isEmpty()

{

if (array[0] == NULL)

{

return false;

}

return true;

}

**//is full**

bool isFull()

{

if (array[size\_array-1] != NULL)

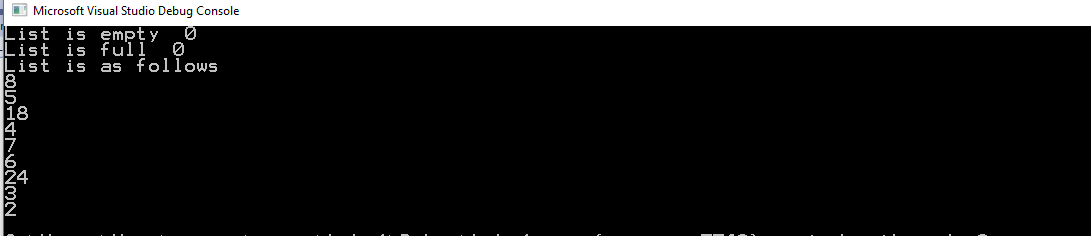
{

return true;

}

return false;

}



**// insertAtposition**

void insertAtposition(int value, int position)

{

if (position > 0)

{

for (int i = (size\_array-1); i >= position; i--)

{

array[i] =array[i-1];

}

array[position-1]=value;

}

if (position >= size\_array)

{

cout << "This position is not available";

}

}

// **sort\_array**

void sort\_array(int arr[size\_array])

{

int temp = 0;

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

{

for (int j = i + 1; j < size\_array; j++)

{

if (arr[i] > arr[j])

{

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

}

}

**//Insert Sorted**

void insertSorted(int value)

{

int temp = 0;

sort\_array(array);

if (value < array[0])

{

for (int i = size\_array - 1; i > 0; i--)

{

{

array[i] = array[i - 1];

}

}

array[0] = value;

}

if (value > array[size\_array - 1])

{

cout << "This could not b inserted" << endl;

}

else

{//some insertion between two numbers

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

{

if ((value > array[i]) && (value < array[i + 1]))

{

for (int j = size\_array - 1; j > i; j--)

{

{

array[j] = array[j - 1];

}

}

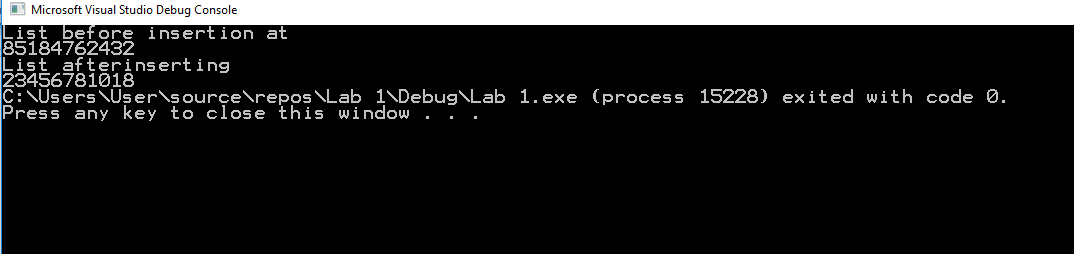
array[i + 1] = value;

}

}

}

}



**// Delete Value**

void Delete\_value(int value)

{

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

{

if (array[i] == value)

{

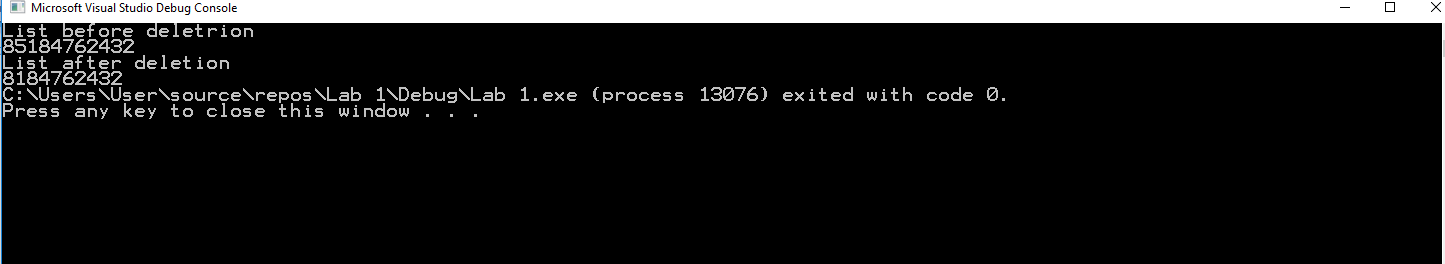
for (i; i < size\_array; i++)

array[i] = array[i + 1];

}

}

}



Delete at Position

void Delete\_at\_position(int position)

{//indexing problem

for (int i = position; i < size\_array; i++)

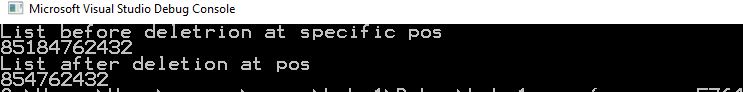
{

array[i-1] = array[i];

}

}

};



**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.

struct DynamicArrayList{

int \*pArray[];

const int size;

pArrayList = new int [size];

int array[size];

bool isEmpty();

bool isFull();

void insertAtposition(int value);

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

}

**isEmpty and isFull() methods**

struct DynamicArrayList

{

int size\_array = 10;

int\* pArrayList = new int[size\_array];

int count = 0;

void enteringdata()

{

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

{

pArrayList[i] = i + 1;

count++;

}

}

bool isEmpty()

{

enteringdata();

if (pArrayList[0] == 0)

return true;

else

return false;

}

bool isFull()

{

enteringdata();

if (count == size\_array)

return true;

else

return false;

}

**OUTPUT**

****

**Delete Value :**

void DeleteValue(int value)

{

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

{

if (value == pArrayList[i])

{

for (i; i <= (size\_array - 1); i++)

{

if (i == (size\_array - 1))

{

pArrayList[i] = 0;

count--;

}

else

pArrayList[i] = pArrayList[i + 1];

}

}

}

if (count <= (1 / 2 \* size\_array))

{

DynamicReduction();

}

}



**Delete at Position**

void DeleteValueAtPosition(int pos)

{

if ((pos > size\_array) || (pos <=1))

{

return;

}

else

{

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

{

if (i == (pos + 1))

{

for (i; i < (size\_array); i++)

{

if (i == (size\_array - 1))

pArrayList[i] = 0;

else

pArrayList[i] = pArrayList[i + 1];

}

count--;

}

}

if (count < 1 / 2 \* size\_array)

{

DynamicReduction();

}

}

}

**Dynamic Reduction**

void DynamicReduction()

{

if (count < 1 / 2 \* (size\_array))

{

new\_size = 1 / 2 \* (size\_array);

int\* tempArray = NULL;

tempArray = new int[new\_size];

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

{

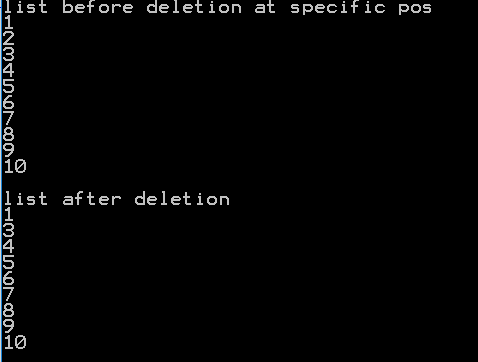
tempArray[i] = pArrayList[i];

}

delete[] pArrayList;

pArrayList = tempArray;

}



**//Insert At Position**

void insertAtposition(int value)

{

if(count== (size\_array-1))

{

DynamicExpansion();

for (int i = 0; i < ((2\*size\_array)-1); i++)

{

if (i > count)

{

pArrayList[i] = value;

count++;

}

}

}

else

{

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

{

if ((i > count) || (pArrayList[i] == 0))

{

pArrayList[i] = value;

count++;

}

}

}

}

**//Sort array**

void sort\_array(int\* ArrayList)

{

int temp = 0;

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

{

for (int j = i + 1; j < size\_array; j++)

{

if (pArrayList[i] == 0)

{

return;

}

else{

if (ArrayList[i] > ArrayList[j])

{

temp = ArrayList[i];

ArrayList[i] = ArrayList[j];

ArrayList[j] = temp;

}

}

}

}

}

**//Insert Sorted**

{

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

{

if (value == pArrayList[i])

{

cout << "already in list" << endl;

return;

}

}

if (count = (size\_array - 1))

{

DynamicExpansion();

int check = 0;

//if user enters 0

if (value == 0)

{

cout << "not a valid value" << endl;

}

else {

for (int i = 0; i < new\_size; i++)

{//checks for greatest value

if (value >= pArrayList[i])

{

check++;

if (check == new\_size)

{

pArrayList[new\_size - 1] = value;

}

}

//checks for a value between two numbers in list

else if ((value > pArrayList[i]) && (value < pArrayList[i + 1]))

{

for (int j = size\_array - 1; j > i; j--)

{

{

pArrayList[j] = pArrayList[j - 1];

}

}

pArrayList[i + 1] = value;

}

}

}

}

}

}

**Dynamic Expansion**

void DynamicExpansion()

{

new\_size = 2 \* size\_array;

int\* tempArray = NULL;

tempArray= new int[new\_size];

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

{

tempArray[i]=pArrayList[i];

cout << tempArray[i];

}

delete pArrayList;

pArrayList = tempArray;

}

**Main Function**

int main()

{

DynamicArrayList d;

cout<< "list is full" << d.isFull() <<endl;

cout<< "list is empty" << d.isEmpty() <<endl;

cout << "list before deletion" << endl;

d.printList();

d.DeleteValue(2);

cout << "list after deletion" << endl;

d.printList();

cout << "list before deletion at specific pos" << endl;

d.printList();

d.DeleteValueAtPosition(2);

cout << "list after deletion" << endl;

d.printList();

d.insertAtPosition();

}