LAB MANUAL

Course: CSC211-Data Structures and Algorithms

Department of Computer Science

Java Learning Procedure

1) Stage J (Journey inside-out the concept)
2) Stage a1 (Apply the learned)
3) Stage v (Verify the accuracy)
4) Stage a2 (Assess your work)

COMSATS University, Islamabad

Table of Contents

Lab #	Topics Covered	Page #
Lab # 01	C++ Revision + Array List	3
Lab # 02	Singly Linked List	10
Lab # 03	Circular Linked list	19
Lab # 04	Doubly Linked List	28
Lab # 05	Stacks	35
Lab # 06	**Lab Sessional 1**	
Lab # 07	Queues	37
Lab # 08	Recursion	39
Lab # 09	Searching	
Lab # 10	Sorting	
Lab # 11	Hashing	
Lab # 12	**Lab Sessional 2**	
Lab # 13	Binary Trees	
Lab # 14	AVL Trees	
Lab # 15	Heaps	
Lab # 16	Graphs	
	Terminal Examination	

LAB # 01

Statement Purpose:

This lab will give you an overview of C++ language.

Activity Outcomes:

This lab teaches you the following topics:

- Basic syntax of C++
- Data types and operators in C++
- Control flow statements in C++
- Arrays and Functions

Instructor Note:

Introduction to Programming in C++

- https://ece.uwaterloo.ca/~dwharder/aads/Tutorial/
- http://www.cplusplus.com

1) Stage J (Journey)

Introduction

C++, as we all know is an extension to C language and was developed by **Bjarne stroustrup** at bell labs. C++ is an intermediate level language, as it comprises a confirmation of both high level and low level language features.

Following features of C++ makes it a stronger language than C,

1. There is Stronger Type Checking in C++.
2. C++ supports and allows user defined operators (i.e Operator Overloading) and function overloading is also supported in it.
3. Exception Handling is there in C++.
4. Inline Functions in C++ instead of Macros in C language. Inline functions make complete function body act like Macro, safely.
5. Variables can be declared anywhere in the program in C++, but must be declared before they are used.

Installing GNU C/C++ Compiler

o install GCC at Windows you need to install MinGW. To install MinGW, go to the MinGW homepage, www.mingw.org, and follow the link to the MinGW download page. Download the latest version of the MinGW installation program which should be named MinGW-<version>.exe.

While installing MinGW, at a minimum, you must install gcc-core, gcc-g++, binutils, and the MinGW runtime, but you may wish to install more.

Add the bin subdirectory of your MinGW installation to your **PATH** environment variable so that you can specify these tools on the command line by their simple names.

When the installation is complete, you will be able to run gcc, g++, ar, ranlib, dlltool, and several other GNU tools from the Windows command line.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

```
// Single line Comment
```

```
/*
* Multiple line
* comment
*/
 1. #include<iostream>
 2. using namespace std;
 3. int main()
 4. {
 5. cout<<"Hello World!";
 6. return 0;
 7. }
```

The output will be “Hello World!”.

Let’s discuss each and every part of the above program.

Line 1:

#include<iostream>

This statements tells the compiler to include iostream file. This file contains pre defined input/output functions that we can use in our program.

Line 2:

using namespace std;

A namespace is like a region, where we have functions, variables etc and their scope is limited to that particular region. Here std is a namespace name, this tells the compiler to look into that particular region for all the variables, functions, etc. I will not discuss this in detail here as it may confuse you. I have covered this topic in a separate tutorial with examples. Just follow the tutorial in the given sequence and you would be fine.

Line 3:

int main()

As the name suggests this is the main function of our program and the execution of program begins with this function, the int here is the return type which indicates to the compiler that this function will return a integer value. That is the main reason we have a return 0 statement at the end of main function.

Line 5:

cout << “Hello World!”;

The `cout` object belongs to the iostream file and the purpose of this object is to display the content between double quotes as it is on the screen. This object can also display the value of variables on screen(don't worry, we will see that in the coming tutorials). The cout object is used together with the << operator to display variables. To combine both text and a variable, separate them with the << operator. Like:

```
int Age = 25;
cout << "I am " << Age << " years old.";
```

Line 6:

return 0;

This statement returns value 0 from the main() function which indicates that the execution of main function is successful. The value 1 represents failed execution.

Activity 2:

Variables are containers for storing data values.

In C++, there are different **types** of variables (defined with different keywords), for example:

- int - stores integers (whole numbers), without decimals, such as 123 or -123
- double - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- string - stores text, such as "Hello World". String values are surrounded by double quotes
- bool - stores values with two states: true or false

Write a program to use primitive data type in C++

```
#include <iostream>
using namespace std;

int main ()
{
  float a=5.5;                  // initial value: 5
  int b(3);                 // initial value: 3
  int c{2};                 // initial value: 2
  float result;               // initial value undetermined

  a = a + b;
  result = a - c;
  cout << result;

  return 0;
}
```

Activity 3:

You have already learned that cout is used to output (print) values. Now we will use cin to get user input. cin is a predefined variable that reads data from the keyboard with the extraction operator (>>). cout is pronounced "see-out". Used for **output**, and uses the insertion operator (<<). cin is pronounced "see-in". Used for **input**, and uses the extraction operator (>>).

Write a program to get the input from user

```
#include <iostream>
using namespace std;

int main ()
{
int x;
cout << "Type a number: "; // Type a number and press enter
cin >> x; // Get user input from the keyboard
cout << "Your number is: " << x; // Display the input value

return 0;

}
```

Activity 4:

Write a program to use string data type in C++

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring;
  mystring = "This is the initial string content";
  cout << mystring << endl;
  mystring = "This is a different string content";
  cout << mystring << endl;
  return 0;
}
```

Activity 5:

Write a program to use arithmetic operators in C++

Solution:

```
#include <iostream>
using namespace std;
```

```
int main ()
{
  int a, b=3;
  a = b;
  a+=2;             // equivalent to a=a+2
  cout << a;
}
```

Activity 6:

Write a program to use relational operators in C++

Solution:

```
#include <iostream>
using namespace std;

int main ()
{
  int a,b,c;

  a=2;
  b=7;
  c = (a>b) ? a : b;

  cout << c << '\n';
}
```

Activity 7:

Write a program to use if-else statement in C++

Solution:

```
#include <iostream>
using namespace std;

int main ()
{
int x;
cin>>x;
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
}
```

Activity 8:

Write a program to use while loop in C++

Solution:

```
#include <iostream>
using namespace std;

int main ()
{
  int n = 10;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "liftoff!\n";
}
```

Activity 9:

Write a program to use do-while loop in C++

Solution:

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str;
  do {
    cout << "Enter text: ";
    getline (cin,str);
    cout << "You entered: " << str << '\n';
  } while (str != "goodbye");
}
```

Activity 10:

Write a program to apply the switch statements.

```
#include <iostream>
using namespace std;

int main ()
{
int day = 4;
switch (day) {
  case 1:
    cout << "Monday";
    break;
  case 2:
    cout << "Tuesday";
    break;
  case 3:
    cout << "Wednesday";
    break;
  case 4:
    cout << "Thursday";
    break;
  case 5:
    cout << "Friday";
    break;
  case 6:
    cout << "Saturday";
    break;
  case 7:
    cout << "Sunday";
    break;

default:

      cout<<"Enter valid day";
}
// Outputs "Thursday" (day 4)

}
```

Activity 11:

Write a program that will add two numbers in a function and call that function in main.

Solution:

```
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
}
```

Activity 12:

Write a program to use the concept of Arrays in C++

Solution:

```
#include <iostream>
using namespace std;

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; ++n )
  {
    result += foo[n];
  }
  cout << result;
  return 0;
}
```

3) Stage V (verify)

Home Activities:

Write the computer program to apply the concepts of Array List. The array list will include the following functions:

1. **Insert the value at end of the list**
2. **Insert the value at start of the list**
3. **Insert the value after specific value**
4. **Insert the value before specific value**
5. **Display the array list**
6. **Delete the value from end of the list**
7. **Delete the value from start of the list**
8. **Delete specific value**

LAB # 02

Statement Purpose:

The purpose of this lab session is to acquire skills in working with singly linked lists.

Activity Outcomes:

This lab teaches you the following topics:

- Creation of singly linked list
- Insertion in singly linked list

- Deletion from singly linked list
- Traversal of all nodes

Instructor Note:

1) Stage J (Journey)

Introduction

A *list* is a finite ordered set of elements of a certain type.
The elements of the list are called *cells* or *nodes*.
A list can be represented *statically*, using arrays or, more often, *dynamically*, by allocating and releasing memory as needed. In the case of static lists, the ordering is given implicitly by the one-dimension array. In the case of dynamic lists, the order of nodes is set by *pointers*. In this case, the cells are allocated dynamically in the heap of the program. Dynamic lists are typically called *linked lists*, and they can be singly- or doubly-linked.
The structure of a node may be:

```
typedef struct nodetype

{

  int key; /* an optional field */

  ... /* other useful data fields */

  struct nodetype *next; /* link to next node */

} NodeT;
```

A singly-linked list may be depicted as in Figure 1.1.

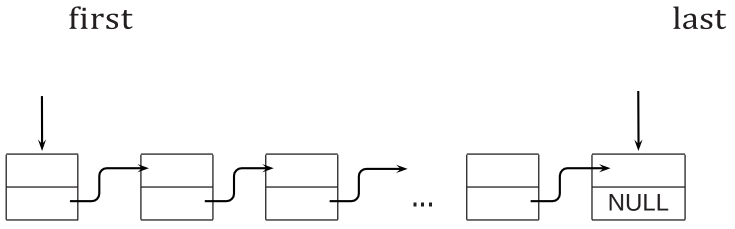


Figure 1.1.: A singly-linked list model.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Creation of linked list

Solution:

Consider the following steps which
apply to *dynamic* lists:

1. Initially, the list is empty. This can be coded by setting the pointers to the first and last cells to the special value
   NULL, i.e. *first* = *NULL*, *last* = *NULL*.

2. Reserve space for a new node to be inserted in the list:

   ```
   /* reserve space */
   p = ( NodeT * )malloc( sizeof( NodeT ));
   ```

   Then place the data into the node addressed by *p*. The implementation depends on the type of data the node holds. For primitive types, you can use an assignment such as $*p = data$.
3. Make the necessary connections:

```
    p->next = NULL;        /* node is appended to the list */
    if ( last != NULL ) /* list is not empty */
last->next = p;
    else
      first = p; /* first node */
    last = p;
```

Activity 2:

Accessing nodes of linked list

Solution:

The nodes of a list may be accessed *sequentially*, gathering useful information as needed. Typically, a part of the infor- mation held at a node is used as a *key* for finding the desired information. The list is scanned linearly, and the node may or may not be present on the list. A function for searching a specific key may contain the following sequence of statements:

```
NodeT *p;
p = first;
while ( p != NULL )
```

```
if ( p->key = givenKey )
{
  return p; /* key found at address p */
}
else
  p = p->next;
return NULL; /* not found */
```

Activity 3:

Insertion in single linked list

Solution:

The node to be inserted may be created as shown at §1.3. We will assume here that the node to insert is pointed to by *p*.

- If the list was empty then there would be only one node, i.e.

  ```
  if ( first == NULL )

  {

    first = p;

    last = p;

    p->next = NULL;

  }
  ```

- If the list was not empty, then insertion follows different patterns depending on the position where the node is to be inserted. Thus, we have the following cases:
  1. Insertion before the first node of the list:

     ```
     if ( first != NULL )

     {

       p->next = first;

       first = p;

     }
     ```

  2. Insertion after the last node of the list (this operation is also called *append*):

     ```
     if ( last != NULL )

     {

       p->n
       ext
       =
     ```

```
    NUL
    L;
    last
    ->n
    ext
    = p;
    last
    = p;

  }
```

3. Insertion before a node given by its *key*. There are two steps to execute:
   a) Search for the node containing the *givenKey*:

```
NodeT *q, *q1;

q1 = NULL; /* initialize */

q = first;

while ( q != NULL )

{

  if ( q->key == givenKey ) break;

  q1 = q;

  q = q->next;

}
```

   b) Insert the node pointed to by *p*, and adjust links:

```
if ( q != NULL )

{

 /* node with key givenKey has address q */

  if ( q == first )

  { /* insert after first node */

    p->next = first;

    first = p;

  }

  else

  {

    q1->next = p;

    p->next = q;

  }

}
```

4. Insertion after a node given by its *key*. Again, there are two steps to execute:
   a) Search for the node containing the *givenKey*:

```
NodeT *q, *q1;
q1 = NULL; /* initialize */
q = first;
while ( q != NULL )
{
  if ( q->key == givenKey ) break;
  q1 = q;
  q = q->next;
}
```

   b) Insert the node pointed to by *p*, and adjust links:

```
if ( q != NULL )
{
  p->next = q->next; /* node with key givenKey has address q */
  q->next = p;
  if ( q == last ) last = p;
}
```

Activity 4:

Deleting a node from single linked list

Solution:

When we are to remove a node from a list, there are some aspects to take into account: (i) list may be empty; (ii) list may contain a single node; (iii) list has more than one node. And, also, deletion of the first, the last or a node given by its key may be required. Thus we have the following cases:

1. Removing the first node of a list

```
NodeT *p;
if ( first != NULL )
{ /* non-empty list */
  p = first;
```

```
    first = first->next;
    free( p ); /* free up memory */
    if ( first == NULL ) /* list is now empty */
last = NULL;
  }
```

2. Removing the last node of a list

```
  NodeT *q, *q1;
  q1 = NULL; /* initialize */
  q = first;
  if ( q != NULL )
  { /* non-empty list */
    while ( q != last )
    { /* advance towards end */
q1 = q;
      q = q->next;
    }
    if ( q == first )
    { /* only one node */
      first = last = NULL;
    }
    else
    { /* more than one node */
      q1->next = NULL;
      last = q1;
    }
    free( q );
  }
```

3. Removing a node given by a *key*

```
  NodeT *q, *q1;
  q1 = NULL; /* initialize */
  q = first;
```

```
        /* search node */
        while ( q != NULL )
        {
          if ( q->key == givenKey ) break;
          q1 = q;
          q = q->next;
        }
        if ( q != NULL )
{   /* found a node with supplied
              key */
          if ( q == first )
          { /* is the first node */
            first = first->next;
            free( q ); /* release memory */
            if ( first == NULL ) last = NULL;
          }
          else
          { /* other than first node */
            q1->next = q->next;
            if ( q == last ) last = q1;
            free( q ); /* release memory */
          }
        }
```

Activity 5:

Complete deletion of single linked list

Solution:

For a complete deletion of a list, we have to remove each node of that list, i.e.

```
NodeT *p;
while ( first != NULL )
{
  p = first;
  first = first->next;
  free( p );
}
last = NULL;
```

3) Stage V (verify)

Home Activities:

1. Write a function that prints all nodes of a linked list in the reverse order.
2. Write a function which reverses the order of the linked list.
3. Write a function which rearranges the linked list by group nodes having even numbered and odd numbered value in their data part.
4. Write a function which takes two values as input from the user and searches them in the list. If both the values are found, your task is to swap both the nodes in which these values are found. Note, that you are not supposed to swap values.

4) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 03

Statement Purpose:

In this lab session we will enhance singly-linked lists with another feature: we'll make them circular. And, as was the case in the previous lab session, we will show how to implement creation, insertion, and removal of nodes.

Activity Outcomes:

This lab teaches you the following topics:

- Creation of circular linked list
- Insertion in circular linked list
- Deletion from circular linked list
- Traversal of all nodes

Instructor Note:

1) Stage J (Journey)

Introduction

A *circular singly linked list* is a singly linked list which has the last element linked to the first element in the list. Being circular it really has no ends; then we'll use only one pointer *pNode* to indicate one element in the list – the newest element. Figure 3.1 show a model of such a list.

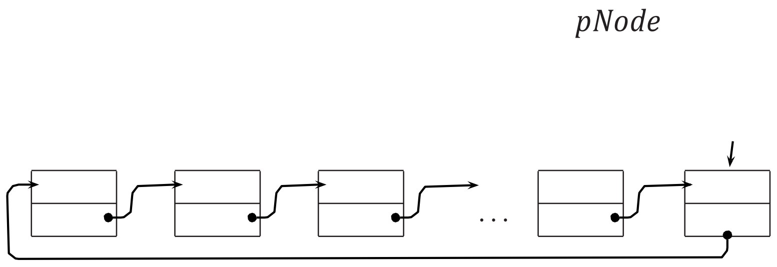


Figure 3.1.: A model of a circular singly-linked list.

The structure of a node may be:

typedef struct *nodetype*

{

  int *key*; */* an optional field */*

  / other useful data fields */*

  struct *nodetype *next*;

  / link to next node */*

} *NodeT*;

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Creation of circular linked list

Solution:

1. Initially, the list is empty, i.e. *pNode* = *NULL*.

2. Generate a node to insert in the list:
   `/* reserve space */`

   p = (*NodeT* *) *malloc*(
   sizeof(*NodeT*)) ; *Then
   read the data into the
   node addressed by p*

3. Link it in the list:
   p->next = NULL;

```
/* node is appended to the list */
if ( pNode == NULL )

{ /*
  empt
  y lis
  t */
  pNode
  = p;
  pNode
  ->nex
  t = p;

}

else

{ /*
  nonempty
  list */
  p->next =
  pNode->nex
  t;
  pNode->nex
  t = p;

  pNode = p;

  /* pNode points
    to the newest
    node in the lis
    t */

}
```

Activity 2:

Accessing nodes of circular linked list

Solution:

The nodes of a list may be accessed sequentially, starting at node *pNode*, as follows:

```
NodeT *p;

p = pNode;
```

```
if ( p != NULL )
  do
  {
    access current node and get data;
    p = p->next;
  }
  while ( p != pNode );
```

Another choice is to look for a key, say *givenKey*. Code for such list scan is given below:

```
NodeT *p;
p = pNode;
if ( p != NULL )
  do
  {
    if ( p->key = givenKey )
    { /* key found at address p */
      return p;
    }
    p = p->next;
  }
  while ( p != NULL );
return NULL; /* not found */
```

Activity 3:

Insertion in circular linked list

Solution:

A node may be inserted *before* a node containing a given key, or *after* it. Both cases imply searching for that key, and, if that key exists, creating the node to insert, and adjusting links accordingly.

Inserting Before a node with key *givenKey*

There are two steps to execute:

1. Find the node with key *givenKey*:

```
NodeT *p, *q, *q1;

q1 = NULL; /* initialize */

q = pNode;

do

{

  q1 = q;

  q = q->next;

  if ( q->key == givenKey ) break;

}

while ( q != pNode );
```

2. Insert the node pointed to by *p*, and adjust links:

```
if ( q->key == givenKey )

{ /* node with key givenKey has address q */

  q1->next = p;

  p->next = q;

}
```

Insertion after a node with key

Again, there are two steps to execute:

1. Find the node with key *givenKey*:*NodeT *p, *q*;

```
q = pNode;

do

{

  if ( q->key == givenKey ) break;

  q = q->next;

}

while ( q != pNode );
```

2. Insert the node pointed to by *p*, and adjust links:

```
if ( q->key == givenKey )
```

```
{ /* node with key givenKey has address q */
  p->next = q->next;
  q->next = p;
}
```

Activity 4:

Deleting a node from circular linked list

Solution:

Again there are two steps to take:

1. Find the node with key *givenKey*:

```
NodeT *p, *q, *q1;
q = pNode;
do
{
  q1 = q;
  q = q->next;
  if ( q->key == givenKey ) break;
}
while ( q != pNode );
```

2. Delete the node pointed to by *q*. If that node is *pNode* then we adjust *pNode* to point to its previous.

```
if ( q->key == givenKey )
{ /* node with key givenKey has address q */
  if ( q == q->next )
  {
    /* list now empty */
  }
  else
  {
    q1->next = q->next;
    if ( q == pNode ) pNode = q1;
  }
  free( q );
```

```
        }
```

Activity 5:

Complete Deletion of Circular Linked list

Solution:

For complete deletion of a list, we have to remove each node of that list, i.e.

```
NodeT *p,  *p1;
p = pNode;
do
{
   p1 = p;
   p = p->next;
   free( p1 );
}
while ( p != pNode );
pNode = NULL;
```

3) Stage V (verify)

Home Activities:

1. Write a function that deletes all those nodes from a linked list which have even/odd numbered value in their data part.
2. Write a function that implements Josephus problem.
3. Write a function that deletes all even positioned nodes from a linked list. Last node should also be deleted if its position is even.

4) Stage a$_2$ (assess)

Lab Assignment and Viva voce

Lab Assignment 1:

Define and implement functions for operating on the data structure given below and the model of figure 3.2.

```
struct circularList
    {
      int length;
      NodeT *first;
    }
```

Operations should be coded as: ins*data*=insert *data* in the list if it is not already there (check the key part of data area for that purpose), del*key*=delete data containing *key* from list (if it is on the list), ist*data*=insert node with *data* as first node, dst=delete first node, prt=print list, fnd*key*=find data node containing *key* in its data area.

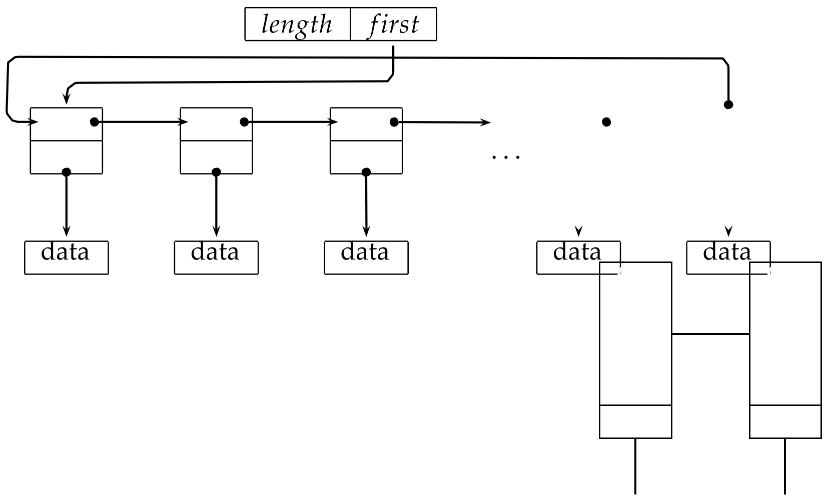


Figure 3.2.: Another model of a circular singly-linked list.

Lab Assignment 2:

Define and implement functions for operating on the data structure given below and the model of figure 3.3.

```
struct circularList
{
   int length;
   NodeT *current;
}
```

The field currentchanges to indicate: the last inserted node if the last operation was insert, the found node for find, the next node for delete, Operations should be coded as: ins*data*=insert a node containing *data* in the list if an element with the same key is not already there, fnd*key*=find data node containing *key* in its data area (check the key part of data area for that purpose), del*key*=delete node with *key* in ints data area from list (if it is on the list), icr*data*=insert after current position, dcr=delete current node, prt=print list.

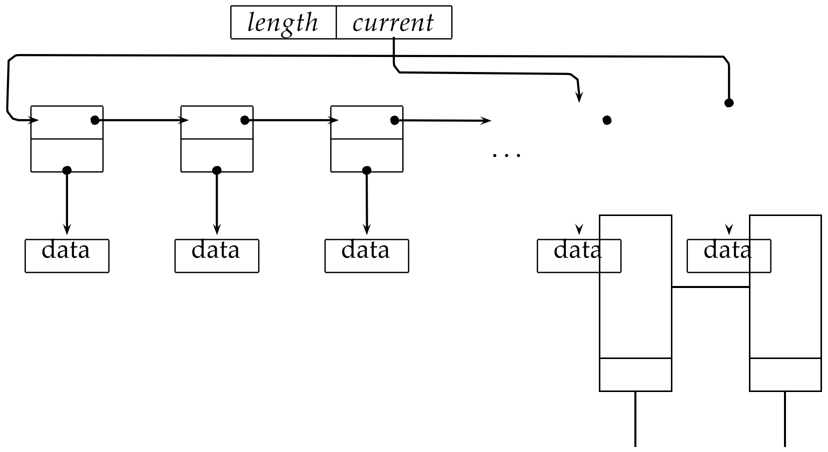


Figure 3.3-..: Yet another model of a circular singly-linked list.

LAB # 04

Statement Purpose:

This lab session is intended to help you develop the operations on doubly-linked lists.

Activity Outcomes:

This lab teaches you the following topics:

- Creation of doubly linked list
- Insertion in doubly linked list
- Deletion from doubly linked list
- Traversal of all nodes

Instructor Note:

1) Stage J (Journey)

Introduction

A *doubly-linked* list is a (dynamically allocated) list where the nodes feature two relationships: *successor* and *predecessor*. A model of such a list is given in figure 4.1. The type of a node in a doubly-linked list may be defined as follows:

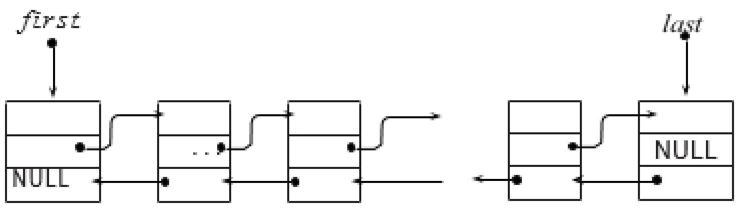


Figure 4.1.: A model of a doubly linked list.

```
typedef struct node_type
{
  KeyT key; /* optional */
  ValueT value;
  /* pointer to next node */
  struct node_type *next;
  /* pointer to previous node */
  struct node_type *prev;
} NodeT;
```

As we have seen when discussing singly-linked lists, the main operations for a doubly-linked list are:

- creating a cell;
- accessing a cell;
- inserting a new cell;
- deleting a cell;
- deleting the whole list.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Creation of doubly linked list

Solution:

In what follows we shall assume that the list is given by a pointer to its header cell, i.e.

```
/* header cell */
struct list_header

{

  NodeT *first;

  NodeT *last;

};

/* list is
   defined as a
   pointer to its
   header */

struct list_header *L;
```

Creating a Doubly Linked List

We shall take the list to be initially empty, i.e.
L->first = L->last = NULL;

After allocating space for a new node and filling in the data (the *value* field), insertion of this node, pointed by *p*, is done as follows:

```
if ( L->first == NULL )

{ /* empty list */

  L->first = L->last = p;

  p->next = p->prev = NULL;

}

else

{ /*
  nonemp
  ty list
  */
  L–>last–
  >next =
  p;
  p–>prev
  = L–>l
  ast;
  L–>last
  = p;
}
```

Activity 2:

Accessing nodes of doubly linked list

Solution:

Stating at a certain position (i.e. at a certain node) we can access a list:

- In sequential forward direction
  ```
  for ( p = L–>first; p != NULL; p = p–>next )
  {
    /* some operation o current cell */
  }
  ```
- In sequential backward direction
  ```
  for ( p = L–>last; p != NULL; p–>p–>prev )
  {
    /* some operation o current cell */
  }
  ```

Activity 3:

Inserting node in doubly linked list

Solution:

We can insert a node before the first node in a list, after the last one, or at a position specified by a given key:

- before the first node:
  ```
  if ( L–>first == NULL )
  { /* empty list */
    L–>first = L–>last = p;
    p_>next = p–>prev = NULL;
  }
  ```

```
else

{ /*
  nonemp
  ty list
  */
  p–>next
  = L–>fir
  st;
  p–>prev
  = NULL;
  L–>first
  –>prev
  = p;
  L–>firs
  t = p;

}
```

- after the last node:

```
if ( L–>first == NULL )

{ /* empty list */

  L–>first = L–>last = p;

  p_>next = p–>prev = NULL;

}

else

{ /*
  nonemp
  ty list
  */
  p–>next
  = NULL;
  p–>prev
  = L–>l
  ast;
  L–>last–
  >next =
  p; L–>l
  ast = p;

}
```

- After a node given by its key:

```
p–>prev = q;

p–>next = q–>next;

if ( q–>next != NULL ) q–>next–>prev = p;q–>next = p;

if ( L–>last == q ) L–>last = p;
```

Here we assumed that the node with the given key is present on list *L* and it we found it and placed a pointer to it in variable *q*.

Activity 4:

Deletion from doubly linked list

Solution:

When deleting a node we meet the following cases:

- Deleting the first node:

```
p = L–>first;
L–>first = L–>first–>next; /* nonempty list assumed */
free( p ); /* release memory taken by node */
if ( L–>first == NULL )
  L–>last == NULL; /* list became empty */
else
  L–>first–>prev = NULL;
```

- Deleting the last node:

```
p = L–>last;
L–>last = L–>last–>prev; /* nonempty list assumed */
if ( L–>last == NULL )
  L–>first = NULL; /* list became empty */
else
  L–>last–>next = NULL;
free( p ); /* release memory taken by node */
```

- Deleting a node given by its key. We will assume that the node of key *givenKey* exists and it is pointed to by *p* (as a result of searching for it)

```
if ( L–>first == p && L–>last == p )
{ /* list has a single node */
  L–>first = NULL;
  L–>last = NULL; /* list became empty */
  free( p );
}
else
if ( p == L–>first )
{ /* deletion of
  first node */
  L–>first = L–>f
  irst–>next; L–>f
```

```
    irst->prev =
    NULL;

    free( p );

  }

  else

  { /* deletion of
    an inner node */
    p->next->prev =
    p->prev;
    p->prev->next =
    p->next;

    free( p );

  }
```

Activity 5:

Deleting complete doubly linked list

Solution:

Deleting a list completely means deleting each of its nodes, one by one.

```
NodeT *p;

while ( L->first != NULL )
{
  p = L->first;

  L->first = L->first->next;

  free( p );

}

L->last = NULL;
```

3) Stage V (verify)

Home Activities:

1. Write a function which reverses the order of a doubly linked list.
2. Write a function which rearranges the linked list by group nodes having even numbered and odd numbered value in their data part.
3. Write a function which takes two values as input from the user and searches them in the list. If both the values are found, your task is to swap both the nodes in which these values are found. Note, that you are not supposed to swap values.

4) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 05

Statement Purpose:

This lab will introduce you the concept of Stack data structure

Activity Outcomes:

This lab teaches you the following topics:

- How to access the top of the stack
- How to push data onto the stack
- How to pop data from the stack

Instructor Note:

1) Stage J (Journey)

Introduction

Stacks

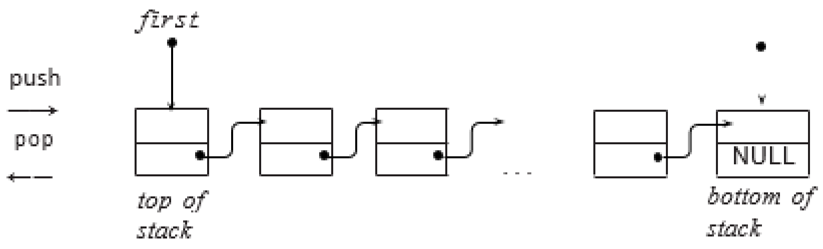
A *stack* is a special case of a singly-linked list which works according to LIFO algorithm. Access is restricted to one end, called the *top* of the stack. A *stack* may be depicted as given in the figure below . Its operations are:

push – push an element onto the top of the stack;

pop – pop an element from the top of the stack;

top – retrieve the element at the top of the stack;

delete – delete the whole stack. This can be done as explained in the previuos paragraph.



2) Stage a1 (apply)

Lab Activities:

1. Write a function that checks whether parentheses in a mathematical expression are balanced or not.
2. Write a function that converts a mathematical expression with no parentheses from infix form to postfix form
3. Write a function that converts an expression from infix form to prefix form.

3) Stage V (verify)

Home Activities:

4. Write a function that converts a mathematical expression **containing parentheses** from infix form to postfix form

4) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 06

Statement Purpose:

This lab will introduce you the concept of Queue data structure

Activity Outcomes:

This lab teaches you the following topics:

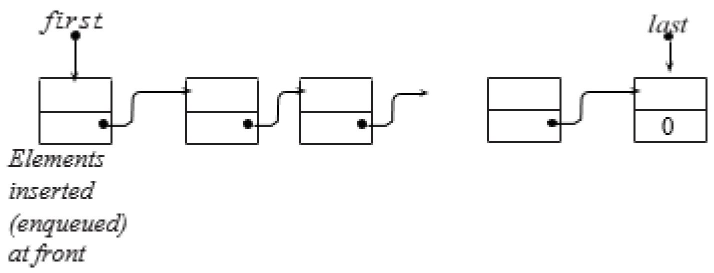
- How to access the front and rear pointers
- How to enqueuer/insert the data
- How to dequeuer/ delete the data

Instructor Note:

`1) Stage J (Journey)

Introduction

A *queue* is also a special case of a singly-linked list, which works according to FIFO[2] algorithm. Of the two ends of the queue, one is designated as the *front* – where elements are extracted (operation called *dequeue*), and another is the *rear*, where elements are inserted (operation called *enqueue*). A *queue* may be depicted as in Figure below:



The main operations are:

Enqueue – place an element at the tail of the queue;
Dequeue – take out an element form the front of the queue;
Delete – delete the whole queue

2) Stage a1 (apply)

Lab Activities:

3) Stage V (verify)

Home Activities:

4) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 07

Statement Purpose:

This lab will introduce you the concept of Recursion

Activity Outcomes:

This lab teaches you the following topics:

- How to formulate programs recursively.
- How to apply the three laws of recursion.
- How to implement the recursive formulation of a problem.
-

Instructor Note:

`1) Stage J (Journey)

Introduction

An algorithm is **recursive** if it calls itself to do part of its work. For this approach to be successful, the "call to itself" must be on a smaller problem then the one originally attempted. In general, a recursive algorithm must have two parts: the **base case**, which handles a simple input that can be solved without resorting to a recursive call, and the recursive part which contains one or more recursive calls to the algorithm where the parameters are in some sense "closer" to the base case than those of the original call.

We divide/decompose the problem into smaller (sub) problems:

- Keep on decomposing until we reach to the smallest sub-problem (base case) for which a solution is known or easy to find
- Then go back in reverse order and build upon the solutions of the sub-problems

A recursive function is defined in terms of base cases and recursive steps.

- In a **base case**, we compute the result immediately given the inputs to the function call.
- In a **recursive step**, we compute the result with the help of one or more recursive calls to this same function, but with the inputs somehow reduced in size or complexity, closer to a base case.

What Happens When a Method is called?

Activation records (AR) will store the following information about the method:

- Local variables of the method.
- Parameters passed to the method.
- Value returned to the calling code (if the method is not a void type).
- The location in the calling code of the instruction to execute after returning from the called method.

Two ways of thinking:

For something simple to start with – let's write a function pow(x, n) that raises x to a natural power of n. In other words, multiplies x by itself n times.

```
pow(2, 2) = 4
pow(2, 3) = 8
pow(2, 4) = 16
```

There are two ways to implement it.

Iterative thinking: the for loop:

```
function pow(x, n) {
  let result = 1;
  // multiply result by x n times in the loop
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  return result;
}
```

Recursive thinking: simplify the task and call self:

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}
```

When pow(x, n) is called, the execution splits into two branches:

```
              if n==1  = x      (Base case)
             /
pow(x, n) =
             \
              else     = x * pow(x, n - 1)      (Recursive call)
```

If n == 1, then everything is trivial. It is called the base of recursion, because it immediately produces the obvious result: pow(x, 1) equals x.

Otherwise, we can represent pow(x, n) as x * pow(x, n - 1). In maths, one would write xn = x * xn-1. This is called a recursive step: we transform the task into a simpler action (multiplication by x) and a simpler call of the same task (powwith lower n). Next steps simplify it further and further until n reaches 1.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a recursive program to calculate factorial of a positive integer

Solution:

Activity 1:

Solution:

Activity 1:

Solution:

Activity 1:

Solution:

Activity 1:

Solution:

Activity 1:

Solution:

3) Stage v (verify)

Home Activities:

4) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 08

Statement Purpose:

This lab will introduce you the concept of Queue data structure

Activity Outcomes:

This lab teaches you the following topics:

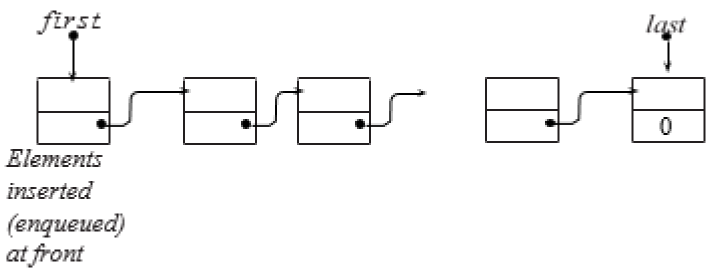
- How to access the front and rear pointers
- How to enqueuer/insert the data
- How to dequeuer/ delete the data

Instructor Note:

`1) Stage J (Journey)

Introduction

A *queue* is also a special case of a singly-linked list, which works according to FIFO[2] algorithm. Of the two ends of the queue, one is designated as the *front* – where elements are extracted (operation called *dequeue*), and another is the *rear*, where elements are inserted (operation called *enqueue*). A *queue* may be depicted as in Figure below:



The main operations are:

Enqueue – place an element at the tail of the queue;
Dequeue – take out an element form the front of the queue;
Delete – delete the whole queue

5) Stage a1 (apply)

Lab Activities:

6) Stage V (verify)

Home Activities:

7) Stage a2 (assess)

Lab Assignment and Viva voce

LAB # 09

Statement Purpose:

This lab will introduce you the concept of Queue data structure

Activity Outcomes:

This lab teaches you the following topics:

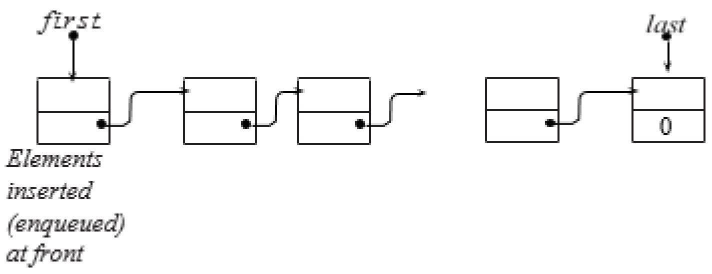
- How to access the front and rear pointers
- How to enqueuer/insert the data
- How to dequeuer/ delete the data

Instructor Note:

`1) Stage J (Journey)

Introduction

A *queue* is also a special case of a singly-linked list, which works according to FIFO[2] algorithm. Of the two ends of the queue, one is designated as the *front* – where elements are extracted (operation called *dequeue*), and another is the *rear*, where elements are inserted (operation called *enqueue*). A *queue* may be depicted as in Figure below:



The main operations are:

Enqueue – place an element at the tail of the queue;
Dequeue – take out an element form the front of the queue;
Delete – delete the whole queue

8) Stage a1 (apply)

Lab Activities:

9) Stage V (verify)

Home Activities:

10) Stage a2 (assess)

Lab Assignment and Viva voce

LAB # 10

Statement Purpose:

This lab will introduce you the concept of Queue data structure

Activity Outcomes:

This lab teaches you the following topics:

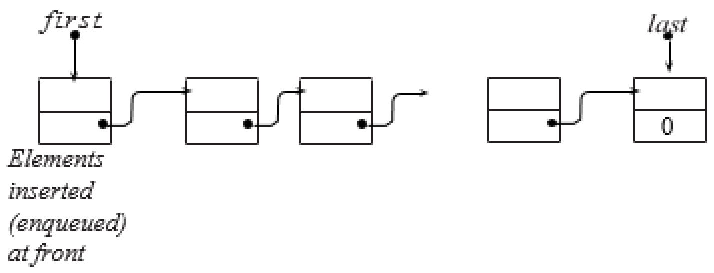
- How to access the front and rear pointers
- How to enqueuer/insert the data
- How to dequeuer/ delete the data

Instructor Note:

`1) Stage J (Journey)

Introduction

A *queue* is also a special case of a singly-linked list, which works according to FIFO[2] algorithm. Of the two ends of the queue, one is designated as the *front* – where elements are extracted (operation called *dequeue*), and another is the *rear*, where elements are inserted (operation called *enqueue*). A *queue* may be depicted as in Figure below:



The main operations are:

Enqueue – place an element at the tail of the queue;
Dequeue – take out an element form the front of the queue;
Delete – delete the whole queue

11) Stage a1 (apply)

Lab Activities:

12) Stage V (verify)

Home Activities:

13) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 11

Statement Purpose:

This lab will introduce you the concept of Queue data structure

Activity Outcomes:

This lab teaches you the following topics:

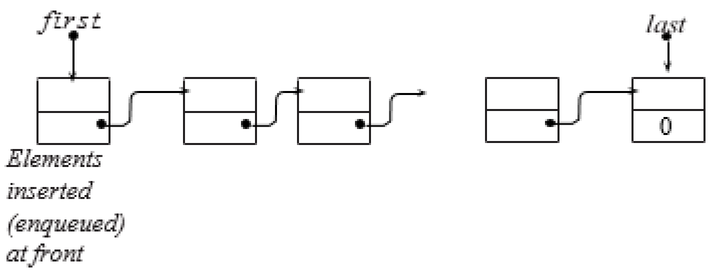
- How to access the front and rear pointers
- How to enqueuer/insert the data
- How to dequeuer/ delete the data

Instructor Note:

`1) Stage J (Journey)

Introduction

A *queue* is also a special case of a singly-linked list, which works according to FIFO[2] algorithm. Of the two ends of the queue, one is designated as the *front* – where elements are extracted (operation called *dequeue*), and another is the *rear*, where elements are inserted (operation called *enqueue*). A *queue* may be depicted as in Figure below:



The main operations are:

Enqueue – place an element at the tail of the queue;
Dequeue – take out an element form the front of the queue;
Delete – delete the whole queue

14) Stage a1 (apply)

Lab Activities:

15) Stage v (verify)

Home Activities:

16) Stage a2 (assess)

Lab Assignment and Viva voce

LAB # 12

Statement Purpose:

This lab will introduce you the concept of Queue data structure

Activity Outcomes:

This lab teaches you the following topics:

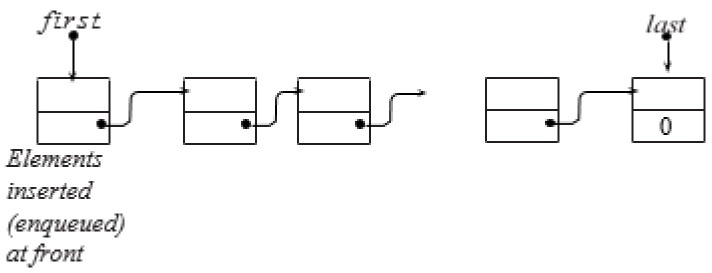
- How to access the front and rear pointers
- How to enqueuer/insert the data
- How to dequeuer/ delete the data

Instructor Note:

`1) Stage J (Journey)

Introduction

A *queue* is also a special case of a singly-linked list, which works according to FIFO[2] algorithm. Of the two ends of the queue, one is designated as the *front* – where elements are extracted (operation called *dequeue*), and another is the *rear*, where elements are inserted (operation called *enqueue*). A *queue* may be depicted as in Figure below:



The main operations are:

Enqueue – place an element at the tail of the queue;
Dequeue – take out an element form the front of the queue;
Delete – delete the whole queue

17) Stage a1 (apply)

Lab Activities:

18) Stage V (verify)

Home Activities:

19) Stage a2 (assess)

Lab Assignment and Viva voce

LAB # 13

Statement Purpose:

This lab will introduce you the concept of Queue data structure

Activity Outcomes:

This lab teaches you the following topics:

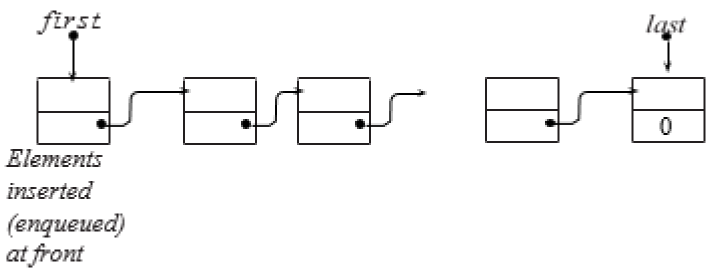
- How to access the front and rear pointers
- How to enqueuer/insert the data
- How to dequeuer/ delete the data

Instructor Note:

`1) Stage J (Journey)

Introduction

A *queue* is also a special case of a singly-linked list, which works according to FIFO[2] algorithm. Of the two ends of the queue, one is designated as the *front* – where elements are extracted (operation called *dequeue*), and another is the *rear*, where elements are inserted (operation called *enqueue*). A *queue* may be depicted as in Figure below:



The main operations are:

Enqueue – place an element at the tail of the queue;
Dequeue – take out an element form the front of the queue;
Delete – delete the whole queue

20) Stage a1 (apply)

Lab Activities:

21) Stage V (verify)

Home Activities:

22) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 14

Statement Purpose:

This lab will introduce you the concept of Queue data structure

Activity Outcomes:

This lab teaches you the following topics:

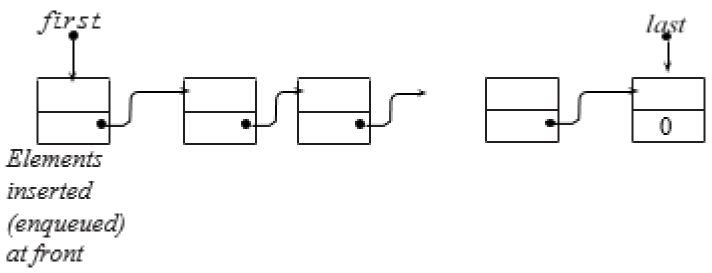
- How to access the front and rear pointers
- How to enqueuer/insert the data
- How to dequeuer/ delete the data

Instructor Note:

`1) Stage J (Journey)

Introduction

A *queue* is also a special case of a singly-linked list, which works according to FIFO[2] algorithm. Of the two ends of the queue, one is designated as the *front* – where elements are extracted (operation called *dequeue*), and another is the *rear*, where elements are inserted (operation called *enqueue*). A *queue* may be depicted as in Figure below:



The main operations are:

Enqueue – place an element at the tail of the queue;
Dequeue – take out an element form the front of the queue;
Delete – delete the whole queue

23) Stage a1 (apply)

Lab Activities:

24) Stage V (verify)

Home Activities:

25) Stage a_2 (assess)

Lab Assignment and Viva voce