Nobody

SINGLY LINKED LIST

Experiment No.:2a

07/08/2012

# AIM:

# Implement Singly Linked List.

# ALGORITHM:

All the linked list algorithms here uses a linked list with nodes with structure as following:

|  |  |
| --- | --- |
| Data | Next |

The members of the structure node are:

Data: Holds the data for the current node.

Next: Holds the linked to the next node.

Here a linked list is represented as a structure as following:

|  |  |
| --- | --- |
| Head | Length |

Head: Head node of the linked list.

Length: Current number of nodes of the linked list initialized at 0.

The function ins() inserts a node at a specific index of a given singly linked list.

Arguments passed:

list: A linked list .

data: The data to be inserted in list.

index: The index where the data is to be inserted in the linked list.

Return value: none.

ins(list, data, index)

{

temp=createnode() /\* The function createnode() is used to allocate a space or a \* node in memory \*/

/\* 'temp', a pointer to node, is used to hold the newly created \* node \*/

temp⇾data=data

temp1=list⇾head /\* 'temp1',a pointer to node, is used to access the list \* sequentially \*/

i=0 /\* 'i' is used to count the index of the node 'temp1' is currently \* holding \*/

while(i<index)

{

temp1=temp1⇾next

i=i+1

}

temp⇾next=temp1

temp1=temp

list⇾length=list⇾length+1

}

The function del() deletes a node at a specific index from a given singly linked list.

Arguments passed:

list: A linkedlist.

index: The index of the node to be deleted in the linked list.

Return value: Returns the deleted data on success and 0 on failure.

del(list, index)

{

temp1=list⇾head /\* 'temp1',a pointer to node is used to access the \* linked list sequentially \*/

i=1 /\* 'i' is used to count the index of the node 'temp1' is current;y \* holding \*/

while(i<index)

{

temp1=temp1⇾next

i=i+1

}

temp=temp1 /\* 'temp', a pointer to mode, holds the node to be deleted \*/

temp1=temp1⇾next

buf=temp⇾data /\* 'buf' holds the data to be returned \*/

freenode(temp) /\* the function freenode() deallocates an allocated space of a \* node in memory \*/

list⇾length=list⇾length-1

return buf

}

The function linear\_search() searches indices all nodes mathing with a given data in a singly linked list using the linear search method.

Arguments passed:

list: A limked list.

data: The data to be searched in the linked list.

Return value: Returns 1 on success and 0 on failure.

linear\_search(list, data)

{

found=0 /\* 'found' is a flag, indicates whether the data is found \*/

temp=list⇾head /\* 'temp' is a pointer to node, used to access the linked list \* sequentially \*/

i=1 /\* 'i' counts the index if the node hold by 'temp' \*/

while(temp ≠ NULL)

{

if(temp⇾data=data)

{

found=1;

print “Data found at position i”

}

temp=temp⇾next

i=i+1

}

if(found=0)

print "Data not found"

return found

}

The function display() prints a singly linked list from a given lower bound to a given upper bound of index.

Arguments passed:

list: A linked list.

lb: Lower bound of the index to printed.

ub: Upper bound of the index to printed.

Return value: None.

display(list, lb, ub)

{

i=1;

temp=list⇾head

while(i ≤ ub AND temp ≠ NULL)

{

if(i ≥ lb)

{

print "temp⇾data”

}

temp=temp⇾next

i=i+1

}

}

# SOURCE CODE:

# #include <iostream>

# #include <cstdlib>

# #include <cstdio>

# #include <vector>

# #include "../myexception.h"

# using namespace std;

# using namespace exception;

# template<class DT> class Node {

# public:

# DT data;

# Node \*next;

# Node(DT, Node\*);

# };

# template<class DT> Node<DT>::Node(DT data, Node\* next=NULL) {

# Node<DT>::data=data;

# Node<DT>::next=next;

# }

# template<class DT> class LinkedList {

# int length;

# Node<DT> \*\*head;

# public:

# LinkedList(int);

# LinkedList(const LinkedList<DT>&);

# ~LinkedList();

# int getLength();

# void insert(int, DT);

# DT remove(int);

# DT& operator[](int);

# DT get(int);

# void set(int, DT);

# void clear();

# void traverse();

# void reverse();

# vector<int> search(DT);

# };

# template<class DT> LinkedList<DT>::LinkedList(int size=0) {

# try{

# LinkedList::head=new Node<DT>\*;

# \*(LinkedList::head)=NULL;

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

# Node<DT> \*temp=\*(LinkedList::head);

# \*(LinkedList::head)=new Node<DT>((DT)0, temp);

# LinkedList::length=0;

# }

# } catch(bad\_alloc) {

# cerr<<"ERROR : bad\_alloc"<<endl;

# }

# }

# template<class DT> LinkedList<DT>::LinkedList(const LinkedList<DT>& l) {

# try{

# LinkedList::head=new Node<DT>\*;

# \*(LinkedList::head)=NULL;

# for(Node<DT> \*temp1=\*(l.head), \*\*temp2=LinkedList::head; temp1!=NULL; temp1=temp1->next, temp2=&((\*temp2)->next))

# \*temp2=new Node<DT>(temp1->data);

# LinkedList::length=l.length;

# } catch(bad\_alloc) {

# cerr<<"ERROR : bad\_alloc"<<endl;

# }

# }

# template<class DT> LinkedList<DT>::~LinkedList() {

# for(Node<DT> \*\*temp=LinkedList::head; \*temp!=NULL;) {

# Node<DT> \*delnode=\*temp;

# \*temp=(\*temp)->next;

# delete delnode;

# }

# LinkedList::length=0;

# delete LinkedList::head;

# }

# template<class DT> int LinkedList<DT>::getLength() {

# return LinkedList::length;

# }

# template<class DT> void LinkedList<DT>::insert(int index, DT data) {

# if(index<0 || index>LinkedList::length)

# throw IndexOutOfBoundsException();

# Node<DT> \*\*temp=LinkedList::head;

# for(int i=0; \*temp!=NULL && i<index; temp=&((\*temp)->next), i++);

# try{

# Node<DT>\* newnode=new Node<DT>(data, \*temp);

# \*temp=newnode;

# LinkedList::length++;

# } catch(bad\_alloc ex) {

# cerr<<"ERROR : bad\_alloc"<<endl;

# }

# }

# template<class DT> DT LinkedList<DT>::remove(int index) {

# if(LinkedList::length==0)

# throw LinkedListEmptyException();

# if(index<0 || index>=LinkedList::length)

# throw IndexOutOfBoundsException();

# Node<DT> \*\*temp=LinkedList::head;

# for(int i=0; \*temp!=NULL && i<index; temp=&((\*temp)->next), i++);

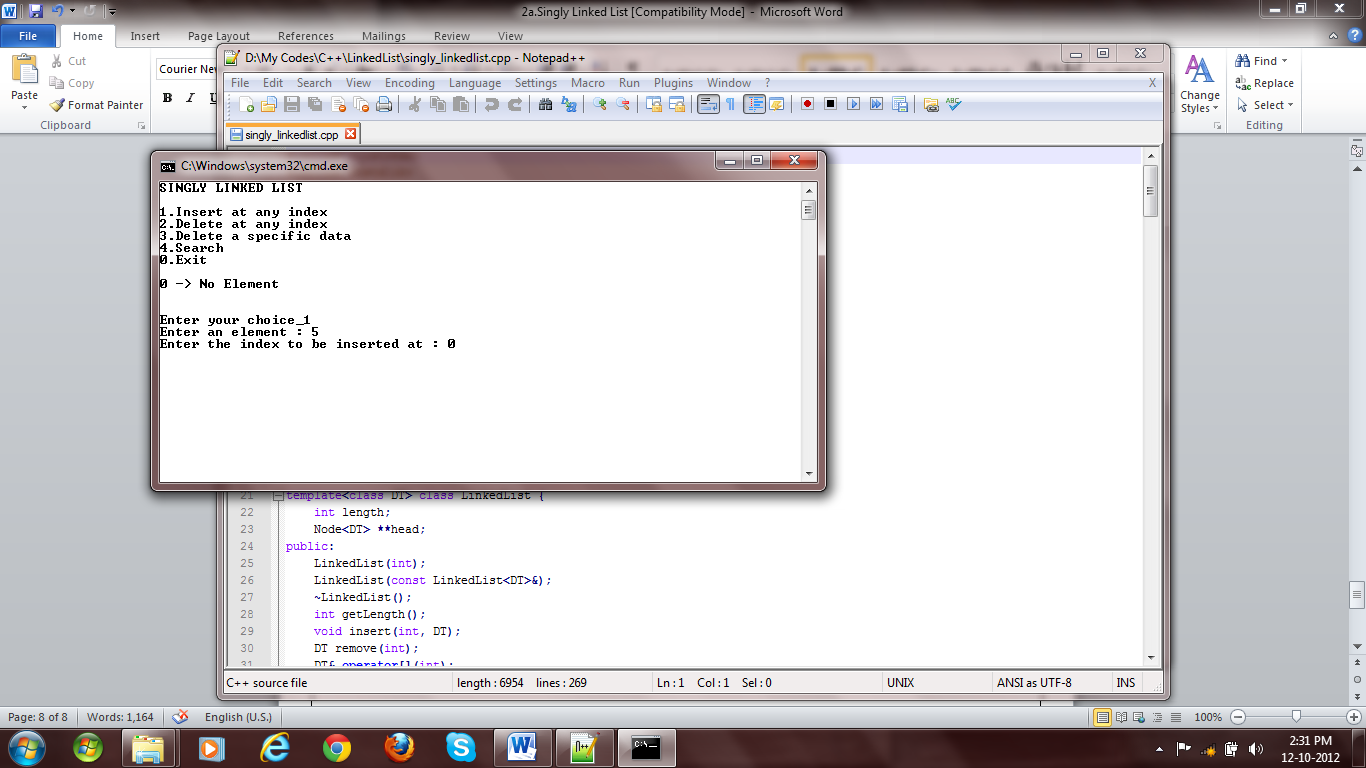
# Node<DT> \*delnode=\*temp;

# \*temp=(\*temp)->next;

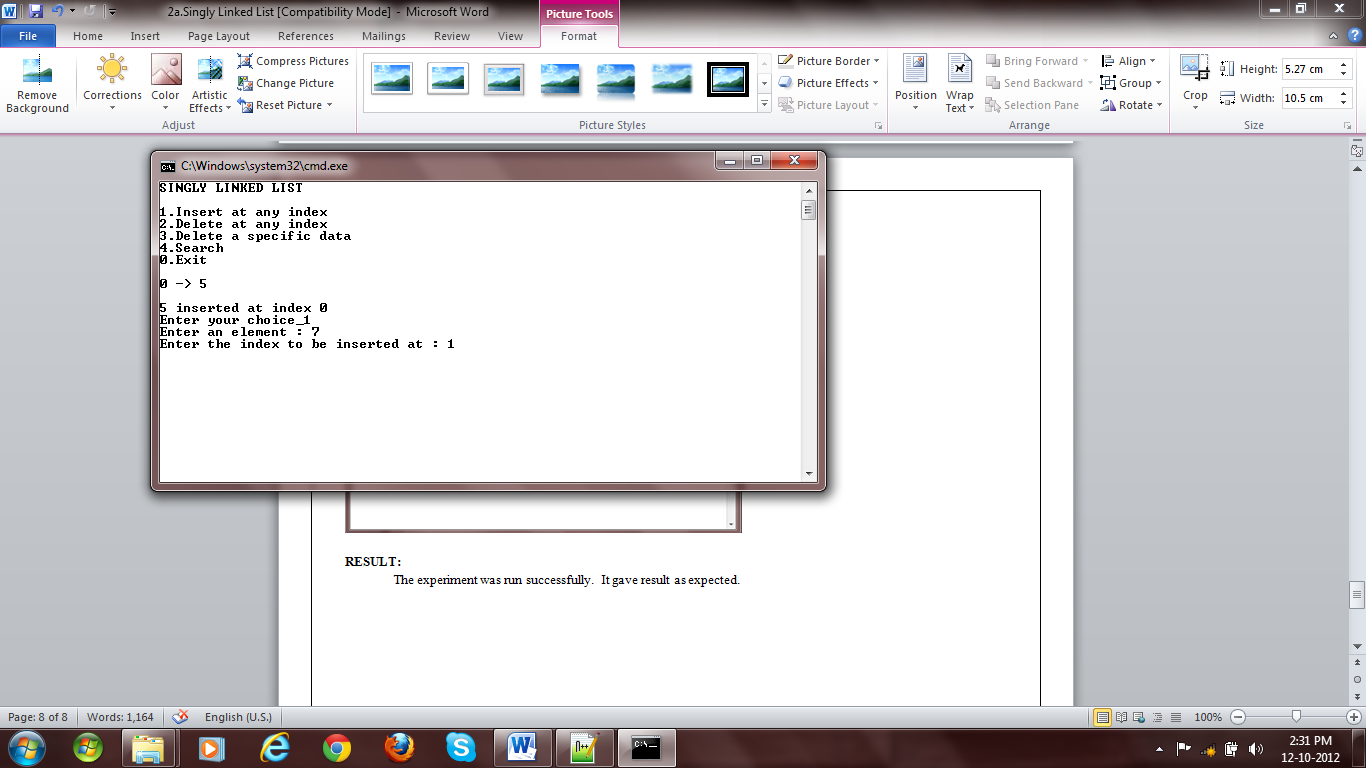
# DT data=delnode->data;

# delete delnode;

# LinkedList::length--;



**Fig 1: Empty Linked List**



**Fig 2:** Insertion

# return data;

# }

# template<class DT> DT& LinkedList<DT>::operator[](int index) {

# if(LinkedList::length==0)

# throw LinkedListEmptyException();

# if(index<0 || index>=LinkedList::length)

# throw IndexOutOfBoundsException();

# Node<DT> \*temp=\*LinkedList::head;

# for(int i=0; temp!=NULL && i<index; temp=temp->next, i++);

# return temp->data;

# }

# template<class DT> DT LinkedList<DT>::get(int index) {

# if(LinkedList::length==0)

# throw LinkedListEmptyException();

# if(index<0 || index>=LinkedList::length)

# throw IndexOutOfBoundsException();

# Node<DT> \*temp=\*LinkedList::head;

# for(int i=0; temp!=NULL && i<index; temp=temp->next, i++);

# return temp->data;

# }

# template<class DT> void LinkedList<DT>::set(int index, DT data) {

# if(LinkedList::length==0)

# throw LinkedListEmptyException();

# if(index<0 || index>=LinkedList::length)

# throw IndexOutOfBoundsException();

# Node<DT> \*temp=\*LinkedList::head;

# for(int i=0; temp!=NULL && i<index; temp=temp->next, i++);

# temp->data=data;

# }

# template<class DT> void LinkedList<DT>::clear() {

# if(LinkedList::length==0)

# throw LinkedListEmptyException();

# for(Node<DT> \*\*temp=LinkedList::head; \*temp!=NULL;) {

# Node<DT> \*delnode=\*temp;

# \*temp=(\*temp)->next;

# delete delnode;

# }

# LinkedList::length=0;

# }

# template<class DT> void LinkedList<DT>::traverse() {

# for(int i=0; i<getLength(); i++)

# cout<<i<<” -> “<<get(i)<<endl;

# }

# template<class DT> void LinkedList<DT>::reverse() {

# Node<DT> \*\*temp=LinkedList::head, \*lastnode=NULL;

# for(int i=0; i<LinkedList::length; i++) {

# Node<DT> \*next=(\*temp)->next;

# (\*temp)->next=lastnode;

# lastnode=\*temp;

# \*temp=next;

# }

# \*LinkedList::head=lastnode;

# }

# template<class DT> ostream& operator<<(ostream& os, LinkedList<DT> l) {

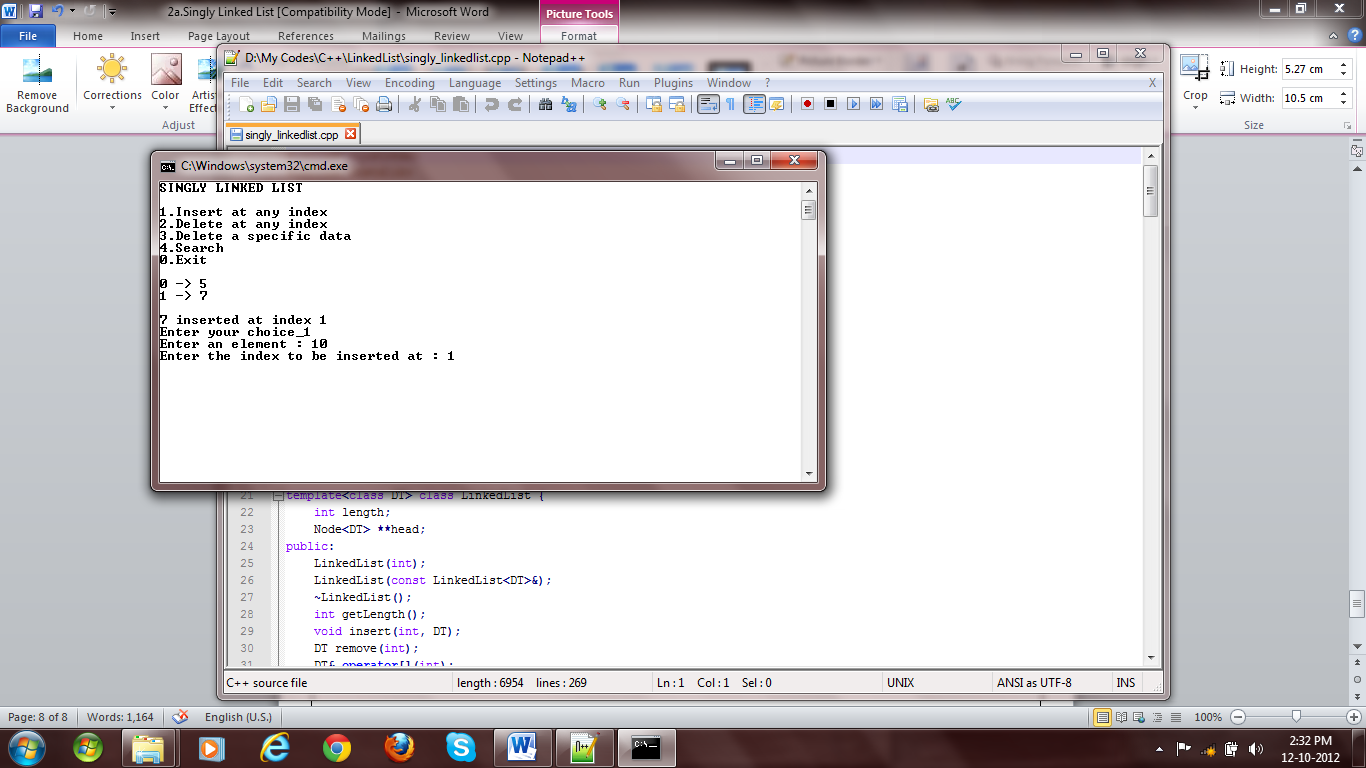


Fig 3: Insertion

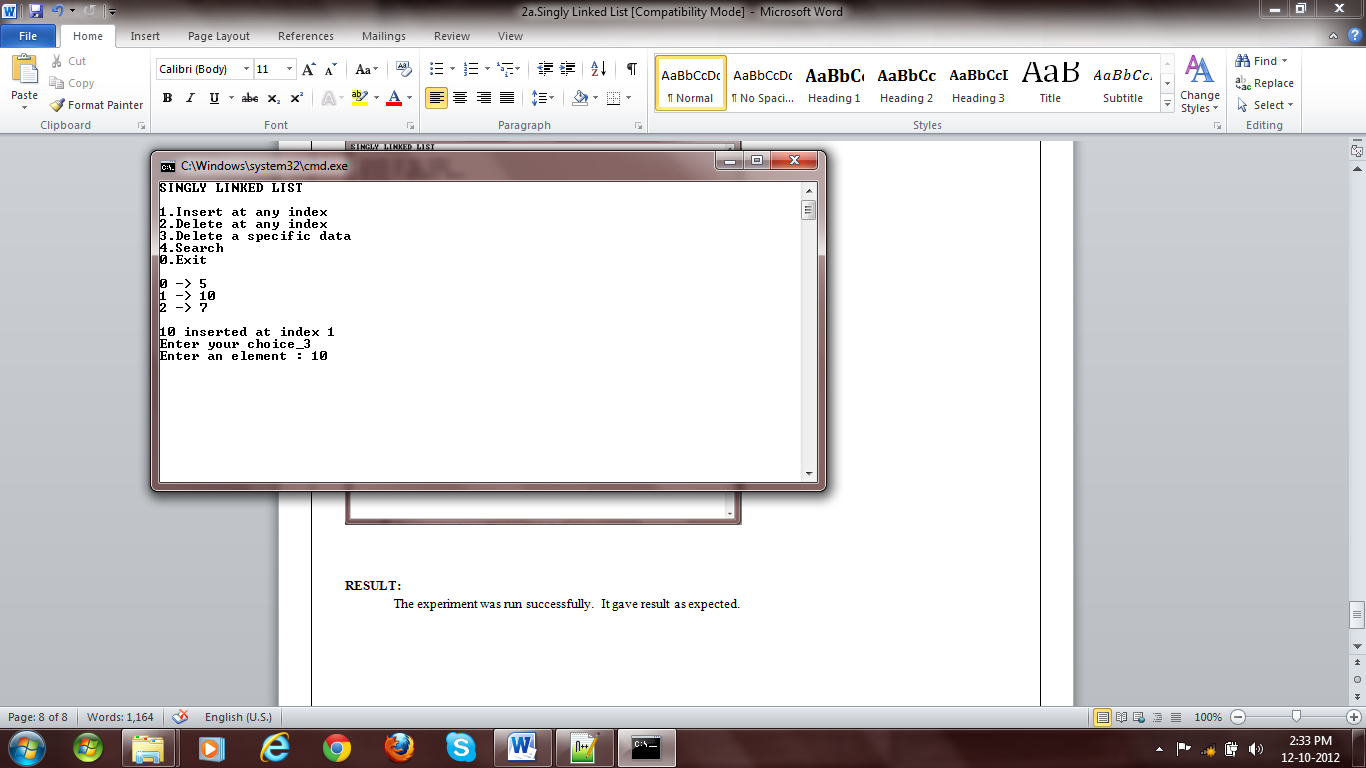


Fig 4: Insertion

# os<<"{";

# for(int i=0; i<l.getLength(); i++)

# os<<(i==0?"":",")<<l[i];

# os<<"}";

# return os;

# }

# template<class DT> vector<int> LinkedList<DT>::search(DT e) {

# vector<int> result;

# Node<DT> \*\*temp=LinkedList::head;

# for(int i=0; \*temp!=NULL; temp=&((\*temp)->next), i++)

# if((\*temp)->data==e)

# result.push\_back(i);

# return result;

# }

# main() {

# LinkedList<int> l(5);

# int choice=1, e, idx;

# vector<int> search\_result;

# while(choice) {

# system("cls");

# cout<<"SINGLY LINKED LIST"<<endl

# <<endl

# <<"1.Insert at any index"<<endl

# <<"2.Delete at any index"<<endl

# <<"3.Delete a specific data"<<endl

# <<"4.Search"<<endl

# <<"5.Display"<<endl

# <<"0.Exit"<<endl

# <<"Enter your choice\_";

# cin>>choice;

# cout<<endl;

# switch(choice) {

# case 0:

# break;

# case 1:

# cout<<"Enter an element : ";

# cin>>e;

# cout<<"Enter the index to be inserted at : ";

# cin>>idx;

# try {

# l.insert(idx, e);

# }catch(IndexOutOfBoundsException ex) {

# cout<<"Error : "<<ex.getMessage()<<endl;

# }

# break;

# case 2:

# cout<<"Enter the index : ";

# cin>>idx;

# try {

# l.remove(idx);

# }catch(IndexOutOfBoundsException ex) {

# cout<<"Error : "<<ex.getMessage()<<endl;

# }catch(LinkedListEmptyException ex) {

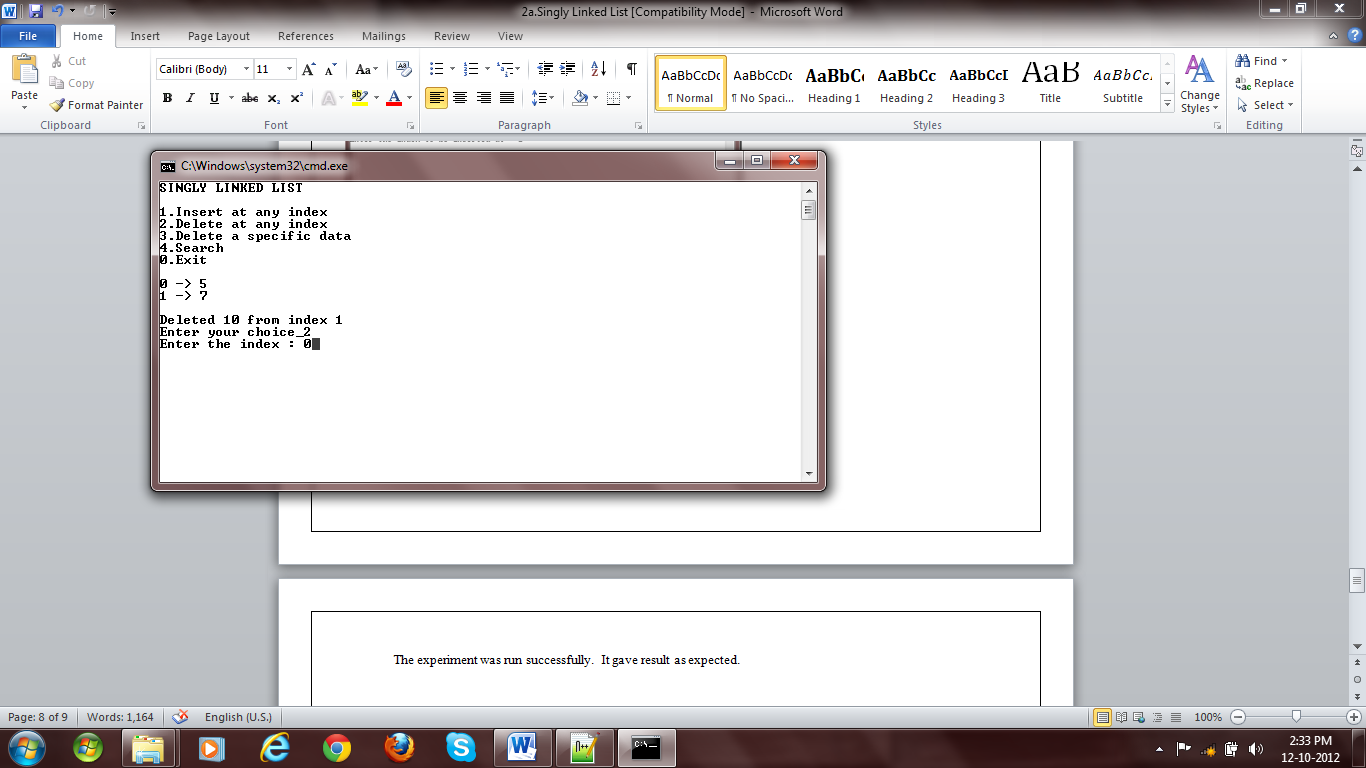


Fig 5: Deletion

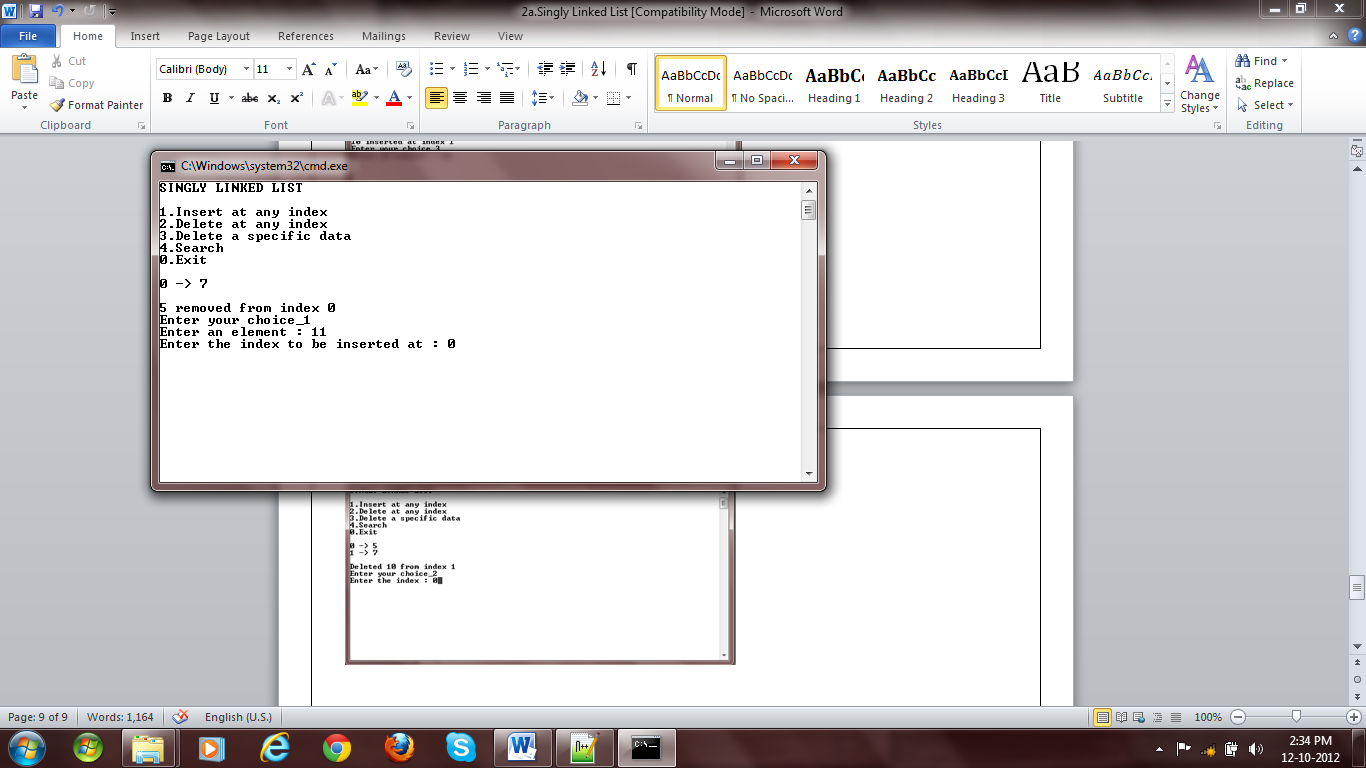


Fig 6: Deletion

# cout<<"Error : "<<ex.getMessage()<<endl;

# }

# break;

# case 3:

# cout<<"Enter an element : ";

# cin>>e;

# search\_result=l.search(e);

# if(search\_result.empty())

# cout<<"Not found"<<endl;

# else if(search\_result.size()==1) {

# l.remove(search\_result[0]);

# cout<<"Deleted at index "<<search\_result[0]<<endl;

# }

# else {

# cout<<"Match found at index ";

# for(int i=0; i<search\_result.size(); i++)

# cout<<search\_result[i]<<",";

# cout<<endl;

# cout<<"Do you want to remove all occurance (y=1/n=0) : "<<endl;

# cin>>choice;

# switch(choice) {

# case 1:

# for(int i=0; i<search\_result.size(); i++)

# l.remove(search\_result[i]-i);

# break;

# case 0:

# cout<<"Enter index : ";

# cin>>idx;

# if(l[idx]==e)

# l.remove(idx);

# else

# cout<<"Wrong index"<<endl;

# break;

# }

# choice=3;

# }

# break;

# case 4:

# cout<<"Enter an element : ";

# cin>>e;

# search\_result=l.search(e);

# if(search\_result.empty())

# cout<<"Not found"<<endl;

# else {

# cout<<"Match found at index ";

# for(int i=0; i<search\_result.size(); i++)

# cout<<search\_result[i]<<",";

# cout<<endl;

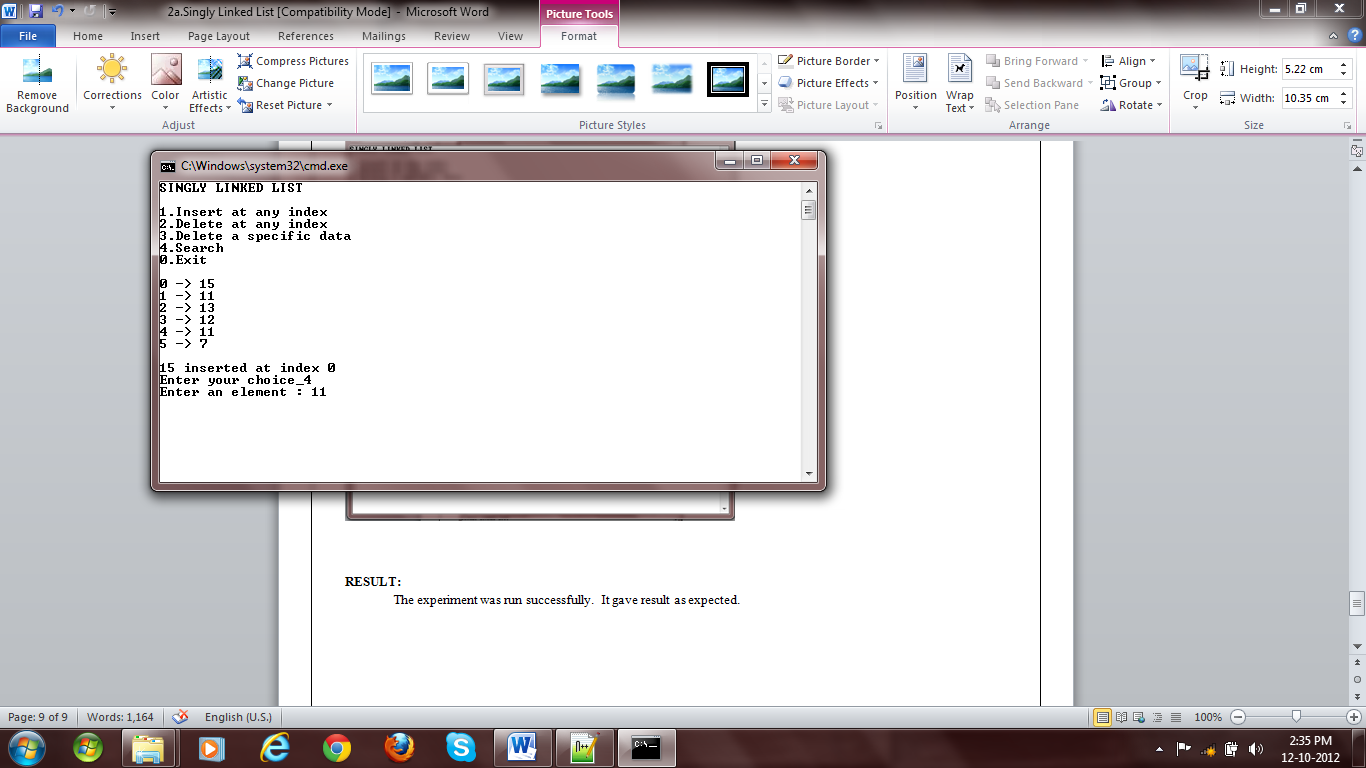
# }

# break;

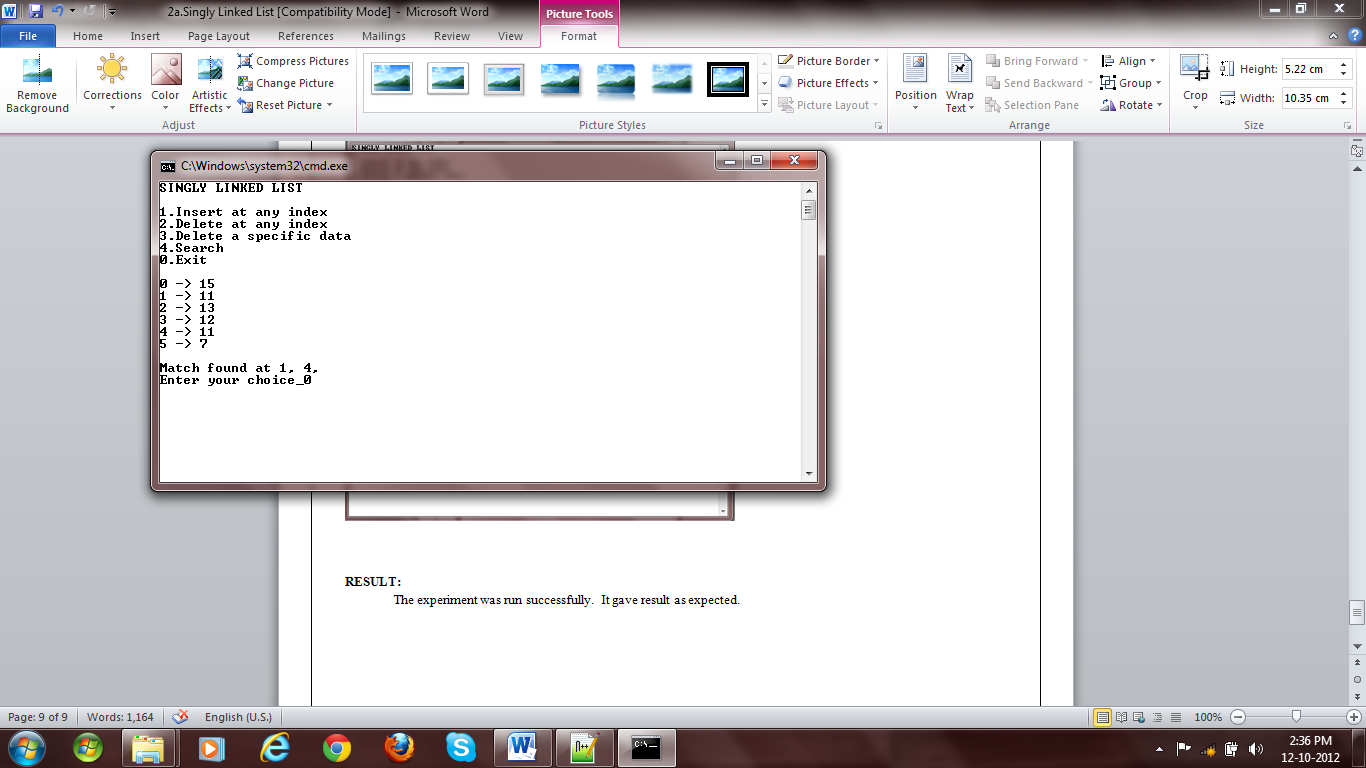
# case 5:

# l.traverse();

# break;



**Fig 7: Search**



**Fig 8. Search**

# default:

# cerr<<"Wrong choice"<<endl;

# }

# cout<<endl

# <<"Press any key..."<<endl;

# getchar();

# getchar();

# }

# }

# RESULT:

The experiment was run successfully. It gave result as expected.