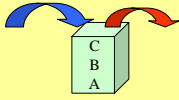



## UCLA Stat 130D Statistical Computing and Visualization in C++

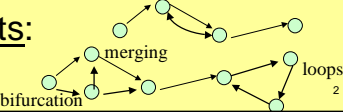
**Instructor: Ivo Dinov, Asst. Prof. in  
Statistics / Neurology**

University of California, Los Angeles, Winter 2004  
[http://www.stat.ucla.edu/~dinov/courses\\_students.html](http://www.stat.ucla.edu/~dinov/courses_students.html)

## Review of Data Structures

**LIFO – stack:** 

**FIFO – Deck/Queue:** 

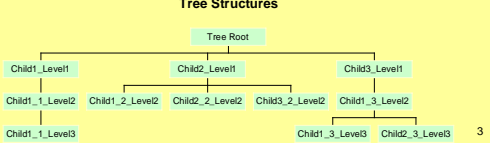
**Linked-Lists:** 

**Graphs:**

## Trees

- Trees are important in design and analysis of algorithms
  - Useful in describing dynamic properties of algorithms
  - We build and use data structures which are direct realizations of trees (*Genotypic evolutionary organization of species*)
  - Ancestor-Descendants data organization (*SuperClass → SubClass*)
  - Hierarchical organization (computer *folder/directory organization*)

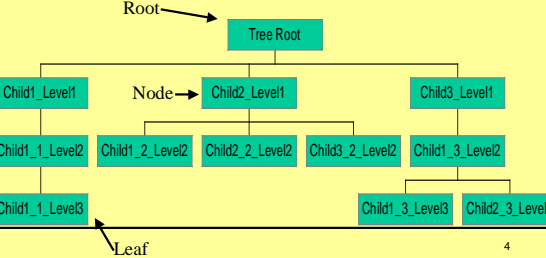
**Tree Structures**



3

## Trees

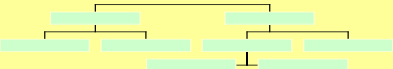
**Tree Structures**



4

## Trees

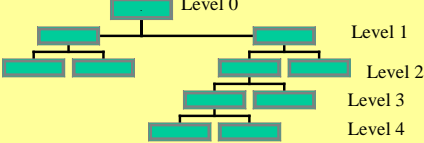
- Tree** – non-empty collection of **vertices** and **edges**, satisfying:
  - A **vertex** is a node that has a name; An **edge** is a connection between two nodes. A **path** in a tree is a collection of distinct vertices in which successive vertices are connected by edges in the tree. **There is precisely one path connecting two nodes in every tree. No loops allowed.**
- Ordered trees** – the order of the children of each node is important.
- Binary trees** – ordered tree where each node has either 2 or no children.



5

## Binary Trees

- Properties of binary trees
  - A **binary tree** with **N internal** (non-terminal) nodes has exactly **(N+1) external** (leaves) nodes.
  - A **binary tree** with **N internal** (non-terminal) nodes has exactly **2N** edges.



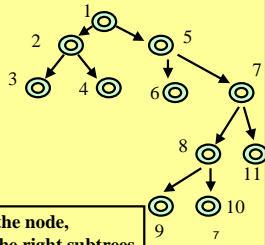
6

## Tree Traversal - Preorder



- Tree **traversal** is the process of systematically processing each node in the tree given a pointer to its root. For **binary trees**, **traversal could be accomplished recursively**:

```
void PreOrderTraverse (link home)
{
    process_current_node();
    if (home==NULL) return;
    PreOrderTraverse(h->left);
    PreOrderTraverse(h->right);
}
```

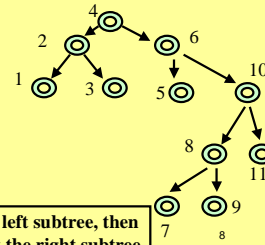


**Preorder traversing** – visit the node, then visit the left and then the right subtrees.

## Binary Tree Traversal - Inorder



```
void InOrderTraverse (link home)
{
    if (home==NULL) return;
    InOrderTraverse(h->left);
    process_current_node();
    InOrderTraverse(h->right);
}
```

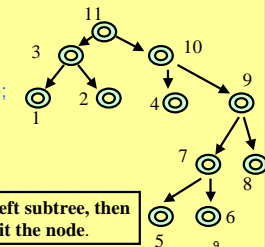


**Inorder traversing** – visit the left subtree, then visit the node and finally visit the right subtree.

## Tree Traversal - Postorder



```
void PostOrderTraverse (link home)
{
    if (home==NULL) return;
    PostOrderTraverse(h->left);
    PostOrderTraverse(h->right);
    process_current_node();
}
```



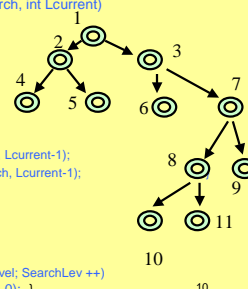
**Postorder traversing** – visit the left subtree, then visit the right subtree, finally visit the node.

**Level-order traversing** – visit all nodes, one level at a time, top-to-bottom, as they appear on the graph.



```
void LevelOrderTraverse (link home, int Lsearch, int Lcurrent)
{
    if (home==NULL) return;
    Node nodeLeft = home->linkLeft;
    Node nodeRight = home->linkRight;
    if (Lcurrent==Lsearch-1)
    {
        process_current_node(nodeLeft);
        process_current_node(nodeRight);
    }
    else if (Lcurrent<Lsearch-1)
    {
        LevelOrderTraverse(nodeLeft, Lsearch, Lcurrent-1);
        LevelOrderTraverse(nodeRight, Lsearch, Lcurrent-1);
    }
    else return;
}

void main ()
{
    // define TREE structure etc.
    process_current_node(Root);
    for (int SearchLev = 1; SearchLev < max_Level; SearchLev++)
    {
        LevelOrderTraverse (root, SearchLev, 0);
    }
}
```



## Tree Traversal

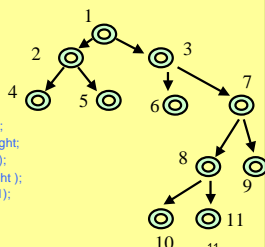


Computing the number of nodes in a tree

```
int numberOfNodes (link h) // Use any traversing algorithm and count Nodes
{
    if (home==NULL) return 0;
    return numberOfNodes(h->l) +
        numberOfNodes(h->r) + 1;
}
```

Computing the height of a tree

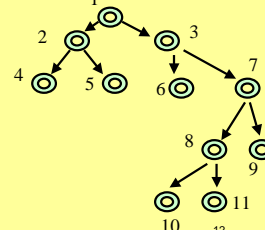
```
int treeHeight(link h)
{
    if (home==NULL) return 0;
    else {
        Node nodeLeft = home->linkLeft;
        Node nodeRight = home->linkRight;
        int leftH = treeHeight(nodeLeft);
        int rightH = treeHeight(nodeRight);
        if (leftH >= rightH) return (leftH+1);
        else return (rightH+1);
    }
}
```



## Brain Tree Example



- Write a program that reads/writes/displays a brain-tree where the nodes are saved in an external file.

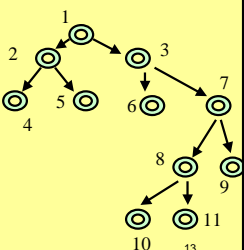


## Tree Class



- Write a prototype for a **Tree Node** class.

```
class TreeNode
{
public:
    TreeNode();
    TreeNode(TreeNode&);
    TreeNode(string);
    TreeNode(char*);
    ~TreeNode();
    TreeNode& getParent();
    void setParent(TreeNode&);
    string getAuxInfo();
    void setAuxInfo(string);
    string getName();
    void setName(string);
    void setName(char*);
private:
    TreeNode* parent;
    string name;
    string* aux_info;
};
```



## Tree Class



- Write a prototype for a **Tree Node** class.

```
TreeNode::TreeNode()
{
    parent = NULL; setName(""); setAuxInfo(""); }

TreeNode::TreeNode(TreeNode& N)
{
    parent = N; setName(""); setAuxInfo(""); }

TreeNode::TreeNode(string s)
{
    parent = NULL; setName(s); setAuxInfo(""); }

TreeNode::TreeNode(char* ch)
{
    parent = NULL; setName(ch); setAuxInfo(""); }

TreeNode::~TreeNode()
{
    delete [] parent; delete [] aux_info; }
```

14

## Tree Class



```
TreeNode::TreeNode& getParent()
{
    return parent; }

TreeNode void setParent(TreeNode& P)
{
    parent = P; }

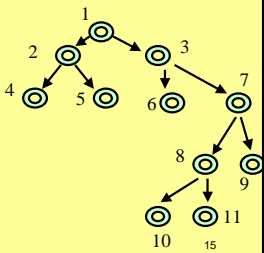
TreeNode::string getName()
{
    return name; }

TreeNode::void setName(string str)
{
    name = str; }

TreeNode::void setName(char* ch)
{
    name = new string(ch); }

TreeNode::string getAuxInfo()
{
    return aux_info; }

TreeNode::void setAuxInfo(string str)
{
    aux_info = str; }
```



## Tree Class

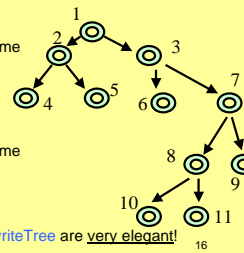


Perhaps we need two additional methods for Reading/Writing The TREE structure out to a file;

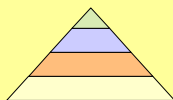
```
class TreeNode
{
public:
    friend int readTree(string);
    // Pre:: string contains a valid file name
    // containing a Tree structure
    // Post:: returns 0 if reading/parsing
    // the file okay, !=0 otherwise

    friend int writeTree(string);
    // Pre:: string contains a valid file name
    // accessible for writing
    // Post:: returns 0 if writing out Tree
    // went okay, !=0 otherwise
}

// Recursive implementations of readTree/writeTree are very elegant!
// Show BrainTree example (java-based)
```



## Sorting Methods



- Sorting Objects
  - Disc/Tape Filename sorting (external, disc block-size)
  - Array sorting (internal, fits in memory, in general)
  - File content sorting (Excel spreadsheet column/row sorting)
- Sorting Efficiency
  - $O(N^2)$  vs.  $O(N \log N)$

17

## “Big-Oh” notation, e.g., $O(n \log(n))$

- We say that the complexity of an algorithm is

$O(f(n))$ , if and only if the ratio

$$g(n) \leq \text{Const} * f(n), \forall n > N_0 \quad \frac{g(n)}{f(n)} \xrightarrow{n \rightarrow \infty} \text{constant}$$

- Where  $g(n)$  is the actual number of operations the algorithm needs to perform to complete the solution to the problem, and  $f(n)$  is some known (say polynomial) function.
- Big-Oh provides a bound for the asymptotic model for the real computational complexity of an algorithm.

18

## “Big-O” notation, e.g., $O(n \log(n))$

- For example, **linear search** for the smallest element in an (unordered) array is  $O(n)$ , since we roughly need to perform  $(n-1)$  comparisons to determine the smallest element

```
min = a[0];
for (int l=1; l<n; l++)
{ if (a[l] < min) min = a[l];
}
```

Actual number of comparisons:  $n-1$

Comparison Poly Factor, in  $O(n)$

Conditional statements are **inexpensive!**

19

## Simple array sorting

Design an array sorting program, any array type with defined (order) operations  $<$  and  $>$

20

```
#include <iostream.h>
#include <stdlib.h>
template <class Item> void swap(Item &A, Item &B)
{ Item t = A; A = B; B = t; }
template <class Item> void complexch(Item &A, Item &B)
{ if (B < A) swap(A, B); }
template <class Item> void sort(Item a[], int l, int r)
{ for (int i = l+1; i <= r; i++)
  for (int j = i+1; j <= r; j++) complexch(a[i], a[j]); }

int main(int argc, char *argv[]) // sorts an array Min → Max
{ int i, N = atoi(argv[1]), sw = atoi(argv[2]);
  int *a = new int[N];
  if (sw) for (i = 0; i < N; i++) a[i] = 1000*(1.0*rand()/RAND_MAX);
  else { N = 0; while (cin >> a[N]) N++; }
  sort(a, 0, N-1);
  for (i = 0; i < N; i++) cout << a[i] << " ";
  cout << endl;
}
```

What's the complexity of the sort method?

```
template <class Item> void sort(Item a[], int l, int r)
{ for (int i = l+1; i <= r; i++)
  for (int j = i+1; j <= r; j++) complexch(a[j], a[i]);
}
```

Complexity of this algorithm is ::

$(N-1)+(N-2) + \dots + 2+1 \sim (N^2)/2$

If  $N=1000$ ,  $(N^2)/2 \sim 1,000,000/2 = 500,000$

Vs. the **most-efficient**  $N \log N$  algorithm:

$N \log N \sim 3,000$

quite a significant difference. This is **NOT an efficient** sorting algorithm.

Too many swap-calls (`complexch( type, type )`)

**Selection Sort** Algorithm - the current element is tested to the smallest element found so far. At most  $(N-1)$  swaps are needed to complete.  $N^2/2$  comparisons and  $N$  exchanges

```
template <class Item>
void selectionSort(Item a[], int l, int r)
{ for (int i = l; i < r; i++)
  { int min = i;
   for (int j = i+1; j <= r; j++)
     if (a[j] < a[min]) min = j;
   swap(a[i], a[min]); // At most N swaps
  }
}
```

**Insertion Sort** Algorithm – consider the elements one at a time and insert them in the list of the already sorted elements by **making room and shifting all larger ones**.

$N^2/4$  comparisons and  $N^2/4$  exchanges

```
template <class Item>
void insertionSort(Item a[], int l, int r)
{ int i;
  for (i = r; i > l; i--) complexch(a[i-1], a[i]);
  // puts smallest element on position 0
  for (i = l+2; i <= r; i++)
  { int j = i; Item v = a[i]; // assignment instead of exchange
    while (v < a[j-1]) { a[j] = a[j-1]; j--; }
    a[j] = v;
  }
}
```

Elements to the left of current index are sorted (min → max). But they are not in final positions

**Bubble Sort Algorithm** – keep passing through the list exchanging adjacent elements which are out of order, continue until the entire list is sorted. (Not very efficient!)

**$N^2/2$  comparisons and  $N^2/2$  exchanges**

```
template <class Item>
void bubbleSort(Item a[], int l, int r)
{ for (int i = l; i < r; i++)
  for (int j = r; j > i; j--)
    compexch(a[j-1], a[j]);
}
```

**Quick Sort Algorithm** – array is processed by a **partition** procedure which puts  $a[i]$  into position for some  $l \leq i \leq r$  and rearranges the other elements so that the recursive calls properly finish the quick-sort. The key is in the **partition** algorithm since it positions  $a[i]$  exactly at the right spot right away:  $a[l] \dots a[i-1] \leq a[i] \leq a[i+1] \dots a[r]$ .

```
template <class Item>
void quickSort(Item a[], int l, int r) // recursive design
{
  if (r <= l) return;
  int i = partition(a, l, r);
  quickSort(a, l, i-1);
  quickSort(a, i+1, r);
}
```

Conditions:

1. Partitioning makes  $a[i]$  be in final position, for some  $i$ .
2. All  $a[l], \dots, a[i-1] \leq a[i]$ .
3. All  $a[i+1], \dots, a[r] \geq a[i]$ .

**Quick Sort – Core partition method**

Computational Complexity:  
1. Most of time:  $N \log(N)$   
2. Worst cases:  $N^2$

```
template <class Item>
int partition(Item a[], int l, int r)
{ int i = l-1, j = r; Item v = a[r];
  for (;;) // infinite loop, terminates when break is called
    // only when the two pointers cross!
    { while (a[++i] < v); // i->, and <-j
      while (v < a[--j]) if (j == l) break;
      if (i >= j) break;
      swap(a[i], a[j]);
    }
  swap(a[i], a[r]);
  return i;
}
```

Strategy:

1. Arbitrary choose  $a[r]$  to be the *partitioning element* to go in its final position.
2. Scan from left until we find element greater than  $a[r]$ . Scan from right to find element less than  $a[r]$ .
3. These two elements are obviously out-of-order – swap them and continue.

**Merging and MergeSort** – combine two ordered arrays into one ordered array.

```
template <class Item>
void merge_AB(Item c[], Item a[], int N, Item b[], int M)
{
  for (int i = 0, j = 0, k = 0; k < N+M; k++)
  {
    if (i == N) { c[k] = b[j++]; continue; }
    if (j == M) { c[k] = a[i++]; continue; }
    c[k] = (a[i] < b[j]) ? a[i++] : b[j++]; // ternary comparison
  }
}
```

## Template Example of Linked List Class

```
#ifndef H_LinkedListType
#define H_LinkedListType

#include <iostream>
using namespace std;

template <class Type>
struct NodeType
{
  Type info;
  NodeType<Type> *link;
};
```

29

## Template Example of Linked List Class

```
template <class Type>
class LinkedListType
{
public:
  // constructors
  LinkedListType();
  LinkedListType(const LinkedListType<Type>& otherList);
  ~LinkedListType();

  // observers
  const LinkedListType<Type>& operator= (const LinkedListType<Type>&);
  bool isEmptyList();
  bool isFullList();
  void Print();
  int Length();
  void RetrieveFirst(Type& firstElement);
  void Search(const Type& searchItem);
```

30

### Template Example of Linked List Class

```
// transformers
void InitializeList();
void DestroyList();
void InsertFirst(const Type& newItem);
void InsertLast(const Type& newItem);
void DeleteNode(const Type& deleteItem);

private:
    NodeType<Type> *first;
    NodeType<Type> *last;
};

#endif
```

31

### Template Example of Linked List Class

```
#include "linkedList.h"
using namespace std;

template<class Type>
LinkedListType<Type>::LinkedListType()
{
    first = NULL;
    last = NULL;
}
```

32

### Template Example of Linked List Class

```
template<class Type>
LinkedListType<Type>::LinkedListType(
    const LinkedListType<Type>& otherList)
{
    NodeType<Type> *newNode;
    NodeType<Type> *current;

    if(otherList.first == NULL)
    {
        first=NULL;
        last=NULL;
    }
}
```

33

### Template Example of Linked List Class

```
else
{
    current = otherList.first;
    first = new NodeType<Type>;
    first->info = current->info;
    first->link = NULL;
    last = first;
    current = current->link;
    while(current != NULL)
    {
        newNode = new NodeType<Type>;
        newNode->info = current->info;
        newNode->link = NULL;
        last->link = newNode;
        last = newNode;
        current = current->link;
    }
}
```

34

### Template Example of Linked List Class

```
template<class Type>
LinkedListType<Type>::~~LinkedListType()
{
    NodeType<Type> *temp;

    while(first != NULL)
    {
        temp = first;
        first = first->link;
        delete temp;
    }

    last = NULL;
}
```

35

### Template Example of Linked List Class

```
template<class Type>
const LinkedListType<Type>&
LinkedListType<Type>::operator=(
    const LinkedListType<Type>& otherList)
{
    NodeType<Type> *newNode;
    NodeType<Type> *current;

    if(this != &otherList)
    {
        if(first != NULL)
            DestroyList();

        if(otherList.first == NULL)
        {
            first = NULL;
            last = NULL;
        }
    }
}
```

36

### Template Example of Linked List Class

```
else
{
    current = otherList.first;
    first = new NodeType<Type>;
    first->info = current->info;
    first->link = NULL;

    last = first;
    current = current->link;
    while(current != NULL)
    {
        newNode = new NodeType<Type>;
        newNode->info = current->info;
        newNode->link = NULL;
        last->link = newNode;
        last = newNode;
        current = current->link;
    }
}
return *this;
}
```

37

### Template Example of Linked List Class

```
template<class Type>
bool LinkedListType<Type>::IsEmptyList()
{
    return(first == NULL);
}

template<class Type>
bool LinkedListType<Type>::IsFullList()
{
    return false;
}
```

38

### Template Example of Linked List Class

```
template<class Type>
void LinkedListType<Type>::Print()
{
    NodeType<Type> *current;

    current = first;
    while(current != NULL)
    {
        cout<<current->info<<" ";
        current = current->link;
    }
}
```

39

### Template Example of Linked List Class

```
template<class Type>
int LinkedListType<Type>::Length()
{
    int count = 0;
    NodeType<Type> *current;

    current = first;
    while (current!= NULL)
    {
        count++;
        current = current->link;
    }

    return count;
}
```

40

### Template Example of Linked List Class

```
template<class Type>
void
LinkedListType<Type>::RetrieveFirst(Type&
    firstElement)
{
    firstElement = first->info;
}
```

41

### Template Example of Linked List Class

```
template<class Type>
void LinkedListType<Type>::Search(const Type& item)
{
    NodeType<Type> *current;
    bool found;

    if(first == NULL)
        cout<<"Cannot search an empty list. "<<endl;
    else {
        current = first;
        found = false;
        while(!found && current != NULL)
            if(current->info == item) found = true;
            else current = current->link;
        if(found)
            cout<<"Item is found in the list."<<endl;
        else
            cout<<"Item is not in the list."<<endl;
    }
}
```

42

### Template Example of Linked List Class

```
template<class Type>
void LinkedListType<Type>::InitializeList()
{
    DestroyList();
}

template<class Type>
void LinkedListType<Type>::DestroyList()
{
    NodeType<Type> *temp;

    while(first != NULL) {
        temp = first;
        first = first->link;
        delete temp;
    }
    last = NULL;
}
```

43

### Template Example of Linked List Class

```
template<class Type>
void LinkedListType<Type>::InsertFirst(const
Type& newItem) {
    NodeType<Type> *newNode;

    newNode = new NodeType<Type>;
    newNode->info = newItem;
    newNode->link = first;
    first = newNode;
    if(last == NULL)
        last = newNode;
}
```

44

### Template Example of Linked List Class

```
template<class Type>
void LinkedListType<Type>::InsertLast(const
Type& newItem)
{
    NodeType<Type> *newNode;

    newNode = new NodeType<Type>;
    newNode->info = newItem;
    newNode->link = NULL;
    if(first == NULL){
        first = newNode;
        last = newNode;
    }
    else
    {
        last->link = newNode;
        last = newNode;
    }
}
```

45

### Template Example of Linked List Class

```
template<class Type>
void LinkedListType<Type>::deleteNode
(const Type & deleteItem)
{
    NodeType<Type> *current;
    NodeType<Type> *trailCurrent;
    bool found;

    if(first == NULL)
        cout<<"Can not delet,empty list.\n";
    else
    {
        if(first->info == deleteItem)
        {
            current = first;
            first = first ->link;
            if(first == NULL) last = NULL;
            delete current;
        }
    }
}
```

46

### Template Example of Linked List Class

```
else
{
    found = false;
    trailCurrent = first;
    current = first->link;

    while((!found) && (current != NULL))
    {
        if(current->info != deleteItem)
        {
            trailCurrent = current;
            current = current-> link;
        }
        else found = true;
    }
}
```

47

### Template Example of Linked List Class

```
if(found) {
    trailCurrent->link = current->link;
    if(last == current)
        last = trailCurrent;
    delete current;
}
else cout<<"Item is not in list.\n";
}
```

48

### Template Example of Linked List Class

```
template<class Type>
const LinkedListType<Type>&
LinkedListType<Type>::operator=(
    const LinkedListType<Type>& otherList)
{
    NodeType<Type> *newNode;
    NodeType<Type> *current;

    if(this != &otherList)
    { if(first != NULL)
      DestroyList();
      if(otherList.first == NULL)
      {
          first = NULL;
          last = NULL;
      }
    }
}
```

49

### Template Example of Linked List Class

```
else
{
    current = otherList.first;
    first = new NodeType<Type>;
    first->info = current->info;
    first->link = NULL;
    last = first;
    current = current->link;
}
```

50

### Template Example of Linked List Class

```
while(current != NULL)
{
    newNode = new NodeType<Type>;
    newNode->info = current->info;
    newNode->link = NULL;
    last->link = newNode;
    last = newNode;
    current = current->link;
}

return *this;
}
```

51

### Template Example of Linked List Class

```
#include <iostream>
#include "linkedlist.h"
using namespace std;

int main(){
    LinkedListType<int> list1, list2;
    int num;

    cout<<"Line 3: Enter numbers ending with -999"
    <<endl;
    cin>>num;

    while(num != -999) {
        list1.InsertLast(num);
        cin>>num;
    }
    cout<<endl;
}
```

52

### Template Example of Linked List Class

```
cout<<"Line 9: List 1: ";
list1.Print();
cout<<endl;

cout<<"Line 12: Length List 1: "
<<list1.Length()<<endl;
list2 = list1;
cout<<"Line 16: List 2: ";
list2.Print();
cout<<endl;

cout<<"Line 17: Length List 2: "
<<list2.Length()
<<endl;

cout<<"Line 18: Enter the number to be "
<<"deleted: ";
cin>>num;
cout<<endl;
```

53

### Template Example of Linked List Class

```
list2.DeleteNode(num);
cout<<"Line 22: After deleting the node, "
<<"List 2: "
<<endl;
list2.Print();
cout<<endl;

cout<<"Line 25: Length List 2: "
<<list2.Length()
<<endl;

return 0;
}
```

54

## Makefile

```
# project linked list
demo : demo.o linkedlist.o
    CC -o demo demo.o linkedlist.o
demo.o : demo.cc linkedlist.h
    CC -c demo.cc
linkedlist.o : linkedlist.cc
linkedlist.h
    CC -c linkedlist
clean :
    rm -rf *.o
```

55

```
Line 3: Enter numbers ending with -999
1 67 23 75 -999

Line 9: List 1: 1 67 23 75
Line 12: Length List 1: 4
Line 16: List 2: 1 67 23 75
Line 17: Length List 2: 4
Line 18: Enter the number to be deleted: 23

Line 22: After deleting the node, List 2:
1 67 75
Line 25: Length List 2: 3
```

56