Review of Data Structures

- **LIFO – stack:**
  - [Diagram of LIFO stack]

- **FIFO – Deck/Queue:**
  - [Diagram of FIFO queue]

- **Linked-Lists:**

- **Graphs:**
  - [Diagram of graph with loops, bifurcation, emerging]

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**

- [Diagram of tree structures]

Trees

- [Diagram of tree root, leaf, node]

- **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.

**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.
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:

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

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

Tree Traversal - Postorder

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

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

Tree Traversal - Inorder

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

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

Tree Traversal

Computing the number of nodes in a tree

```c
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

```c
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);
    }
}
```

Computing the height of a node

```c
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.

```cpp
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;
};
```

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

```cpp
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
  private:
    // 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. \( O(N \log N) \)

"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
  \[
  \lim_{n \to \infty} \frac{g(n)}{f(n)} = \text{constant}
  \]
  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.
**"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 I=1; I<n; I++) 
  { if (a[I] < min) min = a[I]; }
  ```

  Actual number of comparisons: $n-1$;

  $\lim_{n \to \infty} \frac{n-1}{n} = 1$

  Comparison Poly Factor, in $O(n)$

  Conditional statements are inexpensive!

**Simple array sorting**

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

```
#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 compexch(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++) compexch(a[j], a[i]); 
}

int main(int argc, char *argv[])
{ 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;
}
```

**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--) compexch(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;
  }
}
```
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!)

\[ N^2/2 \text{ comparisons and } N^2/2 \text{ exchanges} \]

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--)
    complex(a[j-1], a[j]);
}

Quick Sort Algorithm – array is processed by a partition procedure which puts \( a[i] \) into position for some 1 <= i <= 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: \( a[1], \ldots, a[i-1] <= a[i] <= a[i+1], \ldots, a[r] \).

\[ \text{Conditions:} \]
1. Partitioning makes \( a[I] \) be in final position, for some I.
2. All \( a[l], \ldots, a[I-1] <= a[I] \).
3. All \( a[I+1], \ldots, a[r] >= a[I] \).

Quick Sort – Core partition method

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
      swap(a[i], a[j]);
    if (i >= j) break;
  }
  swap(a[i], a[r]);
  return i;
}

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

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

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

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

template <class Type>
class LinkedListType
{
  public:
    // constructors
    LinkedListType();
    LinkedListType(const LinkedListType<Type>&)
    otherList();
  
    -LinkedListType();
  
    // observers
    const LinkedListType<Type>& operator=
    (const LinkedListType<Type>&&)
    otherList();

    bool isEmptyList();
    bool IsFullList();
    void Print();
    int Length();
    void RetrieveFirst(Type& firstElement);
    void Search(const Type& searchItem);
};
```
Template Example of Linked List Class

```cpp
#include "linkedlist.h"
using namespace std;

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

template<class Type>
LinkedListType<Type>::~LinkedListType()
{
    NodeType<Type> *temp;
    while(first != NULL)
    {
        temp = first;
        first = first->link;
        delete temp;
    }
    last = NULL;
}

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;
            }
        }
        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;
}
```

Template Example of Linked List Class

```cpp
template<class Type>
LinkedListType<Type>::::LinkedListType( const LinkedListType<Type>&& otherList )
{
    NodeType<Type> *newNode;
    NodeType<Type> *current;
    if(current != NULL)
    {
        DestroyList();
        if(otherList.first == NULL)
        {
            first = NULL;
            last = NULL;
        }
        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;
}
```

Template Example of Linked List Class

```cpp
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;
};
```
37

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;

38

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

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

39

template<class Type>
void LinkedListType<Type>::Print()
{
    NodeType<Type> *current;
    current = first;
    while (current != NULL)
    {
        cout << current->info << " ";
        current = current->link;
    }
}

40

template<class Type>
int LinkedListType<Type>::Length()
{
    int count = 0;
    NodeType<Type> *current;
    current = first;
    while (current != NULL)
    {
        count++;
        current = current->link;
    }
    return count;
}

41

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

42

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 (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;
    }
}
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;
}

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;
}

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;
    }
}

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;
        } else { found = false;
            trailCurrent = first;
            current = first->link;
            while(!(found) && (current != NULL)) {
                if(current->info != deleteItem) {
                    trailCurrent = current;
                    current = current->link;
                } else found = true;
            }
            if(found) {
                trailCurrent->link = current->link;
                if(last == current) last = trailCurrent;
                delete current;
            } else cout<<\"Item is not in list.\n\";
        }
    }
}
**Template Example of Linked List Class**

```cpp
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;
        } 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;
}
```

```cpp
#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;
    cout << "Line 9: List 1: " << endl;
    list1.Print();
    cout << endl;
    cout << "Line 12: Length List 1: " << list1.Length() << endl;
    list2 = list1;
    cout << "Line 16: List 2: " << endl;
    list2.Print();
    cout << endl;
    cout << "Line 17: Length List 2: " << list2.Length() << endl;
    cout << "Line 18: Enter the number to be deleted: " << endl;
    cin >> num;
    cout << endl;
    list2.DeleteNode(num);
    cout << "Line 22: After deleting the node," << endl;
    list2.Print();
    cout << endl;
    cout << "Line 25: Length List 2: " << list2.Length() << endl;
    return 0;
}
```
### Makefile

```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
```

### Output

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