Limitation of Arrays And Introduction to Link list

Primitive Data types(SYS Dep)

char	Character or small integer	1 byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
Int	Integer	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
iong int (long)	Long integer	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to

			4294967295
bool	Boolean value. It can take one of two values: true or false	1 byte	true or false
float	Floating point number	4 bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number	8 bytes	+/- 1.7e +/- 308 (~15 digits)

Arrays

Used to store a collection of elements (variables)

```
type array-name[size];
```

Meaning:

This declares a variable called <array-name> which contains <size> elements of type

<type>

The elements of an array can be accessed as:

array-name[0],…array-name[size-1] Example:

```
int a[100]; //a is a list of 100 integers, a[0], a[1],
…a[99]
double b[50];
char c[10];
Examples
```

Drawbacks of Arrays:

1.Fixed Size: Arrays have a predefined size, meaning they cannot grow or shrink dynamically during runtime, which can lead to either memory wastage or overflow.

2.Insertion and Deletion Complexity: Inserting or deleting elements requires shifting elements, resulting in a time complexity of O(n) for these operations.

3.Contiguous Memory Requirement: Arrays require contiguous blocks of memory, which can lead to memory allocation issues, especially for large arrays.

Drawbacks of Arrays:

❖ **Lack of Flexibility:** Arrays do not allow efficient insertion/deletion in the middle, as every operation involves shifting elements.

- ❖ **Homogeneous Data**: Arrays can only store elements of the same data type, limiting flexibility in storing mixed types of data (unless using advanced features like arrays of objects in some languages).

- ❖ **No Built-in Bounds Checking**: Many programming languages do not automatically check array bounds, leading to potential errors like out-of bounds access.

Link list

A linked list is a data structure commonly taught in computer science and programming courses.

It consists of a sequence of nodes, where each node contains data and a reference (or link) to the next node in the sequence.

Linked lists come in various forms.

such as
singly linked lists,
doubly linked lists,
and circular linked lists.

Definition

•A **Singly Linked List** is a linear data structure where each element (node) points to the next one, forming a sequence.
•Each node contains two parts: **data** and a **reference** (or pointer) to the next node.

Characteristics

•Linear structure, unidirectional traversal (can only go forward). •Dynamic in nature (can grow or shrink in size).

Applications

•**Stacks and Queues:** Implemented using Linked Lists for dynamic memory use.

•**Image viewer:** Forward navigation through images.

•**Adjacency List of Graphs:** Used to store edges in graph implementations.

A singly linked list is a collection of nodes, where each node holds two pieces of information:

Data or payload: This is the actual information or value stored in the node.

Reference (or pointer) to the next node: It indicates the location of the next node in the list. This reference connects nodes together, forming a sequential chain.

Nodes: Nodes are the building blocks of a singly linked list.

Each node contains data and a reference to the next node.

The last node in the list typically has a reference pointing to nullptr (or NULL in C++) to signify the end of the list.

Head Pointer: A singly linked list is often managed using a "head" pointer, which points to the first node in the list.

This head pointer allows easy access to the list's elements and facilitates operations like traversal, insertion, and deletion.

Linked Lists Head

∅

- A *linked list* is a series of connected *nodes* • Each node contains at least

- A piece of data (any type)
- Pointer to the next node in the list

• *Head*: pointer to the first node

• The last node points to `NULL`

node

data pointer

Dynamically Allocating Elements

- Allocate elements one at a time as needed, have each element keep track of the *next* element

- Result is referred to as linked list of elements, track next element with a pointer

Array of Elements in Memory
Jane Bob Anne
Linked List
Jane Anne Bob

Linked List Notes

- Need way to indicate end of list **(NULL pointer)**
- Need to know where list starts **(first element)**

• Each element needs pointer to next element **(its link)**

• Need way to allocate new element **(use new)**

• Need way to return element not needed any more **(use free)** • Divide element into **data and pointer**

A Simple Linked List

• We have to make a **structure of**

node. • Define **values includes**

- Name a **link of same data type**

- Example

```
struct node{
      int data;
      node*
next=NULL; };
```

A Simple Linked List Class

- Operations of `List`
  - **`IsEmpty`**: determine whether or not the list is empty
  - **`InsertNode`**: insert a new node at a particular position
  - **`FindNode`**: find a node with a given value
  - **`DeleteNode`**: delete a node with a given value
  - **`DisplayList`**: print all the nodes in the list

Inserting a new node

- Possible cases of **`InsertNode`**
  1. Insert into an empty list
  2. Insert in front
  3. Insert at back
  4. Insert in middle

Insert into an empty list

```
struct node
{
 int x;
 node* next;
};

node* head = nullptr;
void insertIntoEmptyList(Node*& head, int value)  {

Node* newNode = new Node;
newNode->data = value;
newNode->next = nullptr;

head = newNode;
}
```

Insert nodes at front

```
struct node
{
 int x;
 node* next;
};
```

```
node* head = nullptr;
void insertFront(int g) {
 node* temp1 = new node();
temp1->x = g;
 temp1->next = nullptr;
 if (head == nullptr)
{
 head = temp1;
 }
else
{
temp1->next = head;  head = temp1;
 }
}
```

Insert at the end of node

```
void insertEnd(int g)
{
 struct node* temp1 = new node();
temp1->x = g;
 temp1->next = nullptr;

 if (head == nullptr) {
 head = temp1;
 return;
 }

 struct node* pointer = head;  while
(pointer->next != nullptr) {  pointer
= pointer->next;  }

 pointer->next = temp1;
 }
```

```
struct Node {
 int data;
 Node* next;
};
```

```
linked list  Node* current =
head;
 while (current != nullptr) {
```

```
int main() {
 // Create nodes
```

// Define a structure for a node

// Traversal: Display the

```
 Node* head = nullptr;
```

```
Node* second = nullptr;
```

```
 Node* third = nullptr;

// Allocate memory for nodes and populate
data  head = new Node();
 second = new Node();
 third = new Node();

 head->data = 1;
 head->next = second;

 second->data = 2;
 second->next = third;

 third->data = 3;
 third->next = nullptr; // End of the list
    cout << current->data << " -> ";
```

Traverse the nodes

```
current = current->next;
 }
 cout << "nullptr" << std::endl;

 // Deallocate memory (cleanup)
delete head;
 delete second;
 delete third;

 return 0;
}
```

```
struct Node {  int data;
 Node* next; };
Node* head = nullptr;

void traverse() {
 Node* current = head;

 if (current == nullptr) {
 cout << "Empty list" << endl;
 return;
 }

 while (current != nullptr) {
 cout << "Node value is " << current->data
<< endl;  current = current->next;
 }
}
```

Insert in middle

```
bool insertInMiddle(Node*& head, int value, int position)
 {
 if (position <= 0) {
 cout << "Invalid position for insertion." <<endl;
return false;
}

Node* newNode = new Node;
newNode->data = value;
newNode->next = nullptr;

if (position == 1 || head == nullptr) {
// Insert at the beginning or into an empty list
newNode->next = head;
head = newNode;
return true;
}


Node* current = head;
```

```
int currentPosition = 1;

while (currentPosition < position - 1 && current->next != nullptr)
{
current = current->next;
currentPosition++;
}

newNode->next = current->next;
current->next = newNode;
return true;
}
```

Insert in middle

Idea of middle node insertion

- Where to insert
- Pointers

- Make links
- Update links

Insert element at index n

Create a node and set some value and set link to NULL Run loop from 0 t0 n-1 iterations to traverse the list Insert the new node at that place

Track the links

Display the list

$\varnothing$

Head

Insert at 3rd place (Between 2-3)

Head

1 1 2 2 3

3 ∅

Actual code for insert(g,n)

- struct node* newnode = new node();
- newnode->x=g;
- struct node* temp2=head;
- if (n==1){ newnode->next=head;
- head=newnode; return;
- }

- for(int i=1;i<n-1;i++) { temp2=temp2->next; } • newnode->next=temp2->next;
- temp2->next=newnode;

Insert nodes at front (compact)

```
struct node* temp1 = new node();
          temp1->x=g;
          temp1->next=NULL;
          if(head!=NULL)
```

```
temp1->next=head;
head = temp1;
```

Applications of Singly Linked List:

❖ **Dynamic memory allocation** in data structures like stacks and queues.

❖ **Adjacency lists** in graph representation.

❖ **Polynomial manipulations**, where each node holds coefficients and exponents.

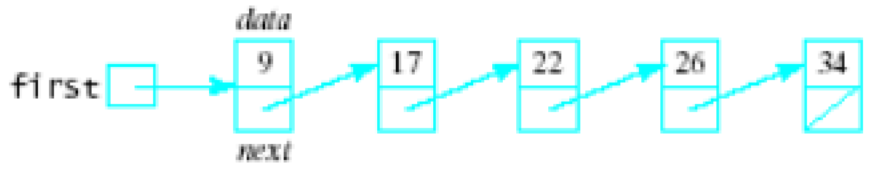
- ❖ **Symbol tables** in compilers.
- ❖ **Navigation through images** in image viewer apps.
- ❖ **Hash tables** using chaining for collision resolution.

Linked List

- Linked list nodes contain
  - Data part – stores an element of the list
  - Next part – stores link/pointer to next element

A Simple Linked List Class

Implementation Overview



(when no next element, null value)

- We use two classes: **`Node`** and **`List`**
- Declare `Node` class for the nodes
  - `data`: `double`-type data in this example
  - `next`: a pointer to the next node in the list

```
class Node {
public:
        double data; // data
        Node* next; // pointer to next
};
```

A Simple Linked List Class

- Declare `List`, which contains
  - `head`: a pointer to the first node in the list.

  Since the list is empty initially, `head` is set to `NULL`

```
class List {
public:
        List(void) { head = NULL; } // constructor
        ~List(void); // destructor

        bool IsEmpty() { return head == NULL; }
        Node* InsertNode(int index, double x);
        int FindNode(double x);
        int DeleteNode(double x);
        void DisplayList(void);
private:
        Node* head;
};
```

A Simple Linked List Class

- Operations of `List`
  - `IsEmpty`: determine whether or not the list is empty • `InsertNode`: insert a new node at a particular position • `FindNode`: find a node with a given value
  - `DeleteNode`: delete a node with a given value
  - `DisplayList`: print all the nodes in the list

Inserting a new node

- Possible cases of `InsertNode`
  1. Insert into an empty list

2. Insert in front

3. Insert at back

4. Insert in middle

- But, in fact, only need to handle two cases
  - Insert as the first node (Case 1 and Case 2)
  - Insert in the middle or at the end of the list (Case 3 and Case 4)

```
Node* List::InsertNode(int index, double x) {
if (index < 0) return NULL;

int currIndex = 1;
Node* currNode = head;
while (currNode && index > currIndex) {
currNode = currNode->next;
currIndex++;
}
if (index > 0 && currNode == NULL) return NULL;

Node* newNode =new Node;
newNode->data = x;
if (index == 0) {
newNode->next = head;
head = newNode;
}
else {
newNode->next = currNode->next;
currNode->next = newNode;
}
return newNode;
}
```

Try to locate `index`'th node. If it doesn't exist, return `NULL`.

Node List::InsertNode(int index, double x) {*

```
if (index < 0) return NULL;

int currIndex = 1;
Node* currNode = head;
while (currNode && index > currIndex) {
currNode = currNode->next;
currIndex++;
}
if (index > 0 && currNode == NULL) return NULL;

Node* newNode =new Node;
newNode->data = x;
if (index == 0) {
newNode->next = head;
head = newNode;
}
else {
newNode->next = currNode->next;
currNode->next = newNode;
}
return newNode;
}
```

Create a new node

```
Node* List::InsertNode(int index, double x) {
if (index < 0) return NULL;

int currIndex = 1;
Node* currNode = head;
while (currNode && index > currIndex) {
currNode = currNode->next;
currIndex++;
}
if (index > 0 && currNode == NULL) return NULL;
```

```
Node* newNode =new Node;
newNode->data = x;
if (index == 0) {
newNode->next = head;
head = newNode;
}
else {
```

Insert as first element head

```
newNode->next = currNode->next;
currNode->next = newNode;
```

```
}
return newNode;
}
```

```
newNode
Node* List::InsertNode(int index, double x) {
if (index < 0) return NULL;

int currIndex = 1;
Node* currNode = head;
while (currNode && index > currIndex) {
currNode = currNode->next;
currIndex++;
}
if (index > 0 && currNode == NULL) return NULL;

Node* newNode =new Node;
newNode->data = x;
if (index == 0) {
newNode->next = head;
head = newNode;
}
else {
```

currNode

Insert after currNode

```
        newNode->next = currNode->next;
        currNode->next = newNode;
    }
    return newNode;
}
```

newNode

Finding a node

- `int FindNode(double x)`
  - Search for a node with the value equal to `x` in the list.
  - If such a node is found, return its position. Otherwise, return 0.

```
int List::FindNode(double x) {
                Node* currNode = head;
       int currIndex = 1;
       while (currNode && currNode->data != x) {
                      currNode = currNode->next;
              currIndex++;
       }
       if (currNode) return currIndex;
       return 0;
}
```

Deleting a node

A0 A1 A2

current

```
current->next = current->next->next;
```

Deleting a node

A0 A1 A2

current

```
Current->next = current->next->next; Memory  leak!
```

Deleting a node

A0 A1 A2

current

```
Node *deletedNode = current->next;
current->next =
current->next->next; delete
deletedNode;
```

Deleting a node

- ***int DeleteNode(double x)***
  - □Delete a node with the value equal to `x` from the list.
  - □If such a node is found, return its position. Otherwise, return 0

    .
- Steps

- □Find the desirable node (similar to `FindNode`)
- □Release the memory occupied by the found node
- □Set the pointer of the predecessor of the found node to the successor of the found node

- ■ Like `InsertNode`, there are two special cases
  - □Delete first node
  - □Delete the node in middle or at the end of the list

```
int List::DeleteNode(double x) {
Node* prevNode = NULL;
Node* currNode = head;
int currIndex = 1;
while (currNode && currNode->data != x) {
prevNode = currNode;
currNode = currNode->next;
currIndex++;
}
if (currNode) {
if (prevNode) {
```

Try to find the node with its value equal to x

```
                                        prevNode->next = currNode->next;
                              delete currNode;
                    }
                    else {
                                                  head = currNode->next;
                              delete currNode;
                    }
                    return currIndex;
          }
          return 0;
}
int List::DeleteNode(double x) {
Node* prevNode = NULL;
Node* currNode = head;
int currIndex = 1;
while (currNode && currNode->data != x) {
prevNode = currNode;
currNode = currNode->next;
currIndex++;
}
if (currNode) {
if (prevNode) {
```

```
prevNode currNode
```

```
                                        prevNode->next = currNode->next;
                              delete currNode;
                    }
                    else {
                                             head = currNode->next;
                              delete currNode;
                    }
                    return currIndex;
          }
          return 0;
}
int List::DeleteNode(double x) {
          Node* prevNode = NULL;
          Node* currNode = head;
          int currIndex = 1;
```

```
            while (currNode && currNode->data != x) {
                        prevNode = currNode;
                        currNode = currNode->next;
                        currIndex++;
            }
            if (currNode) {
                        if (prevNode) {
                                                            prevNode->next = currNode->next;
                                    delete currNode;
                        }
                        else {
                                                                       head = currNode->next;
                                    delete currNode;
                        }
return currIndex;                                    }
}                                                    head currNode
return 0;
```

Printing all the elements

- *void DisplayList(void)*
    - Print the data of all the elements

- Print the number of the nodes in the list

```
void List::DisplayList()
{
 int num = 0;
 Node* currNode = head;
 while (currNode != NULL){
                cout << currNode->data << endl;
                currNode = currNode->next;
                num++;
 }
 cout << "Number of nodes in the list: " << num << endl;
}
```

Destroying the list

- *~List(void)*
  - Use the destructor to release all the memory used by the list.

• Step through the list and delete each node one by one.

```
List::~List(void) {
 Node* currNode = head, *nextNode = NULL;
 while (currNode != NULL)
 {
                          nextNode = currNode->next;
              // destroy the current node
              delete currNode;
              currNode = nextNode;
 }
}
```

Using `List`

```
{
```

```
int main(void)
```

result

List list;

list.InsertNode(0, 7.0); // successful
list.InsertNode(1, 5.0); // successful
list.InsertNode(-1, 5.0); // unsuccessful
list.InsertNode(0, 6.0); // successful
list.InsertNode(8, 4.0); // unsuccessful //

print all the elements
list.DisplayList();

```
Number of nodes in the list: 3 5.0 found
4.5 not found
6
5
Number of nodes in the list: 2
```

if(list.FindNode(5.0) > 0) cout << "5.0 found" << endl;
else cout << "5.0 not found" << endl; if(list.FindNode(4.5) > 0) cout << "4.5 found" << endl;
else cout << "4.5 not found" << endl; list.DeleteNode(7.0);
list.DisplayList();

```
        return 0;
}
```

Linked Lists - Advantages

■Access any item as long as external link to first item maintained

■Insert new item without shifting

■Delete existing item without shifting ■Can expand/contract (flexibile) as necessary

Linked Lists - Disadvantages

■Overhead of links:

- used only internally, pure overhead

- If dynamic, must provide
  - destructor
  - copy constructor
  - assignment operator

- No longer have direct access to each element of the list
  - Many sorting algorithms need direct access
  - Binary search needs direct access
- Access of nth item now less efficient
  - must go through first element, then second, and then third, etc.

Linked Lists - Disadvantages

- List-processing algorithms that require fast access to each element cannot be done as efficiently with linked lists.
- Consider adding an element at the end of the list

Array Linked List

```
a[size++] = value;
```

```
Get a new node;  set data
    part = value
        next part = null_value
    If list is empty
        Set first to point to new node.
    Else
        Traverse list to find last node
    Set next part of last node to
    point to new node.
```

This is the inefficient part

This is the inefficient part

Some Applications?

- A linked list would be a reasonably good choice for implementing any of the following:

1. Applications that have an MRU list (a linked list of file names)
2. The cache in your browser that allows you to hit the BACK button (a linked list of URLs)
3. Undo functionality in Photoshop or Word (a linked list of state)
4. A list in the GPS of the turns along your route

Can we go back in current implementation?

Doubly Linked Lists

next

a b c

prev

head
tail

Consider how hard it is to back up in a singly linked list.

FAST, National University of Computer and Emerging Sciences, Islamabad

Lecture 9 - Linked List Variations

head tail

```
    // Adding first node
 head = new DoubleListNode;
head->next = null;
 head->prev = null;
 tail = head;
```

Inserting into a Doubly Linked List

a c

head tail

current

```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode
```

Inserting into a Doubly Linked List

a c

head tail

current b

```
newNode = new
DoublyLinkedListNode
```

```
newNode->prev = current;
newNode->next                =
current->next;
newNode->prev->next          =
newNode;   newNode->next->prev
= newNode; current = newNode
```

Inserting into a Doubly Linked List

a c

head tail

current b

```
newNode = new
DoublyLinkedListNode
newNode->prev = current;
newNode->next                     =
current->next;
newNode->prev->next               =
newNode;   newNode->next->prev
= newNode; current = newNode
```

Inserting into a Doubly Linked List

a c

head tail

current b

```
newNode = new
```

```
DoublyLinkedListNode
newNode->prev = current;
newNode->next                    =
current->next;
newNode->prev->next              =
newNode;   newNode->next->prev
= newNode; current = newNode
```

Inserting into a Doubly Linked List

a c

head
b tail

current

```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
                             // current->next=newNode;
newNode->next->prev = newNode;
current = newNode
```

Inserting into a Doubly Linked List

a c

head tail

current b

```
newNode = new
DoublyLinkedListNode
newNode->prev = current;
newNode->next                  =
current->next;
newNode->prev->next              =
newNode;    newNode->next->prev
= newNode; current = newNode
```

Inserting into a Doubly Linked List

a c

head

b tail

current

```
newNode = new DoublyLinkedListNode
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode
```

Deleting an element from a double linked list

a c

head b

current

```
oldNode=current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
delete oldNode;
```

Deleting an element from a double linked list

a c

head b

current oldNode

```
oldNode=current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
current = oldNode->prev;
delete oldNode;
```

Deleting an element from a double

linked list

a c

head b

current oldNode

```
oldNode=current;
oldNode->prev->next  =  oldNode->next;
oldNode->next->prev  =  oldNode->prev;
current = oldNode->prev;
delete oldNode;
```

Deleting an element from a double linked list

a c

head b

current oldNode

```
oldNode=current;
oldNode->prev->next  =  oldNode->next;
oldNode->next->prev  =  oldNode->prev;
current = oldNode->prev;
```

```
delete oldNode;
```

Deleting an element from a double linked list

a c

head b

current oldNode

```
oldNode=current;
oldNode->prev->next = oldNode->next;
oldNode->next->prev = oldNode->prev;
```

```
current = oldNode->prev;
delete oldNode;
```

Deleting an element from a double linked list

a c

head

current

```
oldNode=current;
oldNode->prev->next = oldNode->next;
```

```
oldNode->next->prev  =  oldNode->prev;
current = oldNode->prev;
delete oldNode;
```

Circular Linked lists

a b c d

first

Sorted Linked List

A sorted linked list is one in which items are in sorted order. It can be derived from a list class.

What is improved?

InsertNode operation? No

DeleteNode & SearchNode operations? Yes

Thank you