17.1 Introduction to Linked Lists

- A data structure representing a list
- A series of nodes chained together in sequence
  - Each node points to one other node.
- A separate pointer (the head) points to the first item in the list.
- The last element points to nothing (NULL)

```
list
head
```

Introduction to Linked Lists

- The nodes are dynamically allocated
  - The list grows and shrinks as nodes are added/removed.
- Linked lists can easily **insert** a node between other nodes
- Linked lists can easily **delete** a node from between other nodes

```
list head
```

Node Organization

- Each node contains:
  - data field – may be organized as a structure, an object, etc.
  - a pointer – that can point to another node

```
data
pointer
```
**Linked List Organization**

- A linked list contains 0 or more nodes.
- The list head is a pointer that points to the first node.
- Each node points to the next node in the list.
- The last node points to NULL (address 0).

![Linked List Diagram]

**Empty List**

- An empty list contains 0 nodes.
- The list head points to NULL (address 0).
- (There are no nodes, it’s empty)

![Empty List Diagram]

---

**Declaring the Linked List data type**

- We will be defining a class for a linked list data type that can store values of type double.
- The data type will describe the values (the lists) and operations over those values.
- In order to define the values we must:
  - define a data type for the nodes
  - define a pointer variable (head) that points to the first node in the list.

**Declaring the Node data type**

- Use a struct for the node type

```cpp
struct ListNode {
    double value;
    ListNode *next;
};
```

- (this is just a data type, no variables declared)
- next can hold the address of a ListNode.
  - it can also be NULL
  - “self-referential data structure”
Defining the Linked List variable

- Define a pointer for the head of the list:
  ```
  ListNode *head = NULL;
  ```
- It must be initialized to NULL to signify the end of the list.
- Now we have an empty linked list:
  ```
  head
  ```

Using NULL

- Equivalent to address 0
- Used to specify end of the list
- Use ONE of the following for NULL:
  ```
  #include <iostream>
  #include <cstddef>
  ```
- to test a pointer for NULL (these are equivalent):
  ```
  while (p) ...  <==>  while (p != NULL) ...
  if (!p) ...  <==>  if (p == NULL) ...
  ```

17.2 Linked List operations

- Basic operations:
  - create a new, empty list
  - append a node to the end of the list
  - insert a node within the list
  - delete a node
  - display the linked list
  - delete/destroy the list

Linked List class declaration

```
#include <cstddef>   // for NULL
using namespace std;

class NumberList
{
  private:
    struct ListNode    // the node data type
    {
      double value;           // data
      struct ListNode *next;  // ptr to next node
    }; ListNode *head;    // the list head

  public:
    NumberList();      // creates an empty list
    NumberList();
    void appendNode(double);
    void insertNode(double);
    void deleteNode(double);
    void displayList();
}; NumberList.h
```
Operation: **Create** the empty list

- Constructor: sets up empty list

```cpp
#include "NumberList.h"

NumberList::NumberList()
{
    head = NULL;
}
```

appendNode: **append** node to end of list

- appendNode: adds new node to end of list
- Algorithm:

  Create a new node and store the data in it
  If the list has no nodes (it's empty)
  Make head point to the new node.
  Else
  Find the last node in the list
  Make the last node point to the new node

appendNode: find last elem

- How to find the last node in the list?
- Algorithm:

  Make a pointer `p` point to the first element
  while (the node `p` points to) is not pointing to NULL
  make `p` point to (the node `p` points to) is pointing to

- In C++:

  ```cpp
  ListNode *p = head;
  while ((p->next) != NULL)    // while p not NULL
      p = (p->next);            // p = p->next;
  ```

  `<==>`

  ```cpp
  ListNode *p = head;
  while (p->next)              // while p not NULL
      p = p->next;             // p = p->next;
  ```

  `p=p->next` is like `i++`
Driver to demo NumberList

- ListDriver.cpp version 1 (no output)

```cpp
#include "NumberList.h"

int main() {
    // Define the list
    NumberList list;
    // Append some values to the list
    list.appendNode(2.5);
    list.appendNode(7.9);
    list.appendNode(12.6);
    return 0;
}
```

Traversing a Linked List

- Visit each node in a linked list, to display contents, sum data, test data, etc.
- Basic process:

  set a pointer to point to what head points to while pointer is not NULL process data of current node go to the next node by setting the pointer to the pointer field of the current node end while

Operation: **display** the list

```cpp
void NumberList::displayList() {
    ListNode *nodePtr;  //ptr to traverse the list
    // start nodePtr at the head of the list
    nodePtr = head;
    // while nodePtr pts to something (not NULL), continue while (nodePtr)
    {
        //Display the value in the current node
        cout << nodePtr->value << endl;
        //Move to the next node
        nodePtr = nodePtr->next;
    }
}
```

Driver to demo NumberList

- ListDriver.cpp version 2

```cpp
#include "NumberList.h"

int main() {
    // Define the list
    NumberList list;
    // Append some values to the list
    list.appendNode(2.5);
    list.appendNode(7.9);
    list.appendNode(12.6);
    // Display the values in the list.
    list.displayList();
    return 0;
}
```

Output:

2.5
7.9
12.6
**Operation:**

**delete** a node from the list

- **deleteNode**: removes node from list, and deletes (deallocates) the removed node.
- Requires two pointers:
  - one to point to the node to be deleted
  - one to point to the node **before** the node to be deleted.

```cpp
string name1 = "Steve Jobs";
cout << "Name" << name1 << endl;
```

**Deleting a node**

- Change the pointer of the previous node to point to the node **after** the one to be deleted.

```cpp
previousNode->next = nodePtr->next;
```

- Now just "delete" the nodePtr node

**Delete Node Algorithm**

- **After the node is deleted:**

```cpp
delete nodePtr;
```

- **Delete the node containing num**

  If list is empty, exit  //nothing to delete
  If first node contains num  //no previous node
    make p point to first node
    make head point to second node
    delete p
  else
    use p to traverse the list, until it points to num or NULL
    --as p is advancing, make n point to the node before if (p is not NULL)
    make n’s node point to what p’s node points to
    delete p’s node
deleteNode code

```cpp
void NumberList::deleteNode(double num) {
    // empty list, exit
    if (head==NULL)
        return;

    ListNode *nodePtr;      // to traverse the list
    // check if first node is num
    if (head->value == num) {
        nodePtr = head;
        head = nodePtr->next;
        delete nodePtr;
    }
    // else is continued on next slide
    else {
        ListNode *previousNode;  // trailing node pointer
        // initialize traversal ptr to first node
        nodePtr = head;
        // skip nodes not equal to num, stop at last
        while (nodePtr && nodePtr->value != num) {
            previousNode = nodePtr;   // save it!
            nodePtr = nodePtr->next;  // advance it
        }
        // nodePtr not null: num is found, set links + delete
        if (nodePtr) {
            previousNode->next = nodePtr->next;
            delete nodePtr;
        }
        // else: end of list, num not found in list
    }
}
```

Driver to demo NumberList

```cpp```
// set up the list
NumberList list;
list.appendNode(2.5);
list.appendNode(7.9);
list.appendNode(12.6);
list.displayList();
cout << endl << "remove 7.9:" << endl;
list.deleteNode(7.9);
list.displayList();
cout << endl << "remove 8.9: " << endl;
list.deleteNode(8.9);
list.displayList();
cout << endl << "remove 2.5: " << endl;
list.deleteNode(2.5);
list.displayList();
```

Operation: destroy a List

- The destructor must “delete” (deallocate) all nodes used in the list
- To do this, use list traversal to visit each node
- For each node,
  - save the address of the next node in a pointer
  - delete the node
**Destructor**

- `~NumberList`: deallocates all the remaining nodes

```cpp
NumberList::~NumberList() {
    ListNode *nodePtr; // traversal ptr
    ListNode *nextNode; // saves the next node
    nodePtr = head;    // start at head of list
    while (nodePtr) {
        nextNode = nodePtr->next; // save the next
        delete nodePtr;            // delete current
        nodePtr = nextNode;        // advance ptr
    }
}
```

**Operation:**

**insert** a node into a linked list

- Requires two pointers to traverse the list:
  - pointer to point to the node after the insertion point
  - pointer to point to node before point of insertion
- New node is inserted between the nodes pointed at by these pointers

- The before and after pointers move in tandem as the list is traversed to find the insertion point
  - Like delete

**Inserting a Node into a Linked List**

- New node created, new position located:

```
list head
-- 5 -- 13 -- 19 -- NULL

newNode
```

- Insertion completed:

```
list head
-- 17 -- NULL

newNode
```

```cpp
string name1 = "Steve Jobs";
cout << "Name" << name1 << endl;
```
Insert Node Algorithm

Create the new node, store the data in it
If list is empty,
   make head point to new node, new node to null
else
   use p to traverse the list,
      until it points to node after insertion point or NULL
      --as p is advancing, make n point to the node before
   if p points to first node (n is null)
      make head point to new node
      new node to p's node
   else
      make n's node point to new node
      make new node point to p's node

Note that in the insertNode implementation that follows, the insertion point is immediately before the first node in the list that has a value greater than the value being inserted.

This works very nicely if the list is already sorted and you want to maintain the sort order.

Another way to specify the insertion point is to have insertNode take a second argument that is the index of the node after the insertion point.

In this case you can use a count-controlled loop to advance the pointer(s) through the list.

insertNode code

```cpp
void NumberList::insertNode(double num) {
    ListNode *newNode;       // ptr to new node
    ListNode *nodePtr;       // ptr to traverse list
    ListNode *previousNode;  // node previous to nodePtr

    //allocate new node
    newNode = new ListNode;
    newNode->value = num;

    // empty list, insert at front
    if (!head) {
        head = newNode;
        newNode->next = NULL;
    }

    //else is on the next slide . . .
```

insertNode code cont.

```cpp
else {
    // initialize the two traversal ptrs
    nodePtr = head;
    previousNode = NULL;

    // skip all nodes less than num
    while (nodePtr && nodePtr->value < num) {
        previousNode = nodePtr;   // save
        nodePtr = nodePtr->next;  // advance
    }

    if (previousNode == NULL) { //insert before first
        head = newNode;
        newNode->next = nodePtr;
    }
    else {                    //insert after previousNode
        previousNode->next = newNode;
        newNode->next = nodePtr;
    }
}
```
Driver to demo NumberList

```cpp
int main() {
    // set up the list
    NumberList list;
    list.appendNode(2.5);
    list.appendNode(7.9);
    list.appendNode(12.6);
    list.insertNode(10.5);
    list.displayList();
    return 0;
}
```

Advantages of linked lists over arrays

- A linked list can easily grow or shrink in size.
  - The programmer doesn’t need to know how many nodes will be in the list.
- Nodes are created in memory as they are needed.
- When a node is inserted into or deleted from a linked list, none of the other nodes have to be moved.

Advantages of arrays over linked lists

- Arrays allow random access to elements: array[i]
  - linked lists allow only sequential access to elements (must traverse list to get to i’th element).
- Arrays do not require extra storage for “links”
  - linked lists are impractical for lists of characters or booleans (pointer value is bigger than data value).

Exercise: find four errors

```cpp
int main() {
    struct node {
        int data;
        node * next;
    }

    // create empty list
    node * list;

    // insert six nodes at front of list
    node *n;
    for (int i=0;i<=5;i++) {
        n = new node;
        n->data = i;
        n->next = list;
    }

    // print list
    n = list;
    while (n) {
        cout << n->data << "  ";
        n = n->next;
    }
    cout << endl;
    return 0;
}
```