UTM
UNIVERSITI TEKNOLOGI MALAYSIA

DATA STRUCTURE & ALGORITHM

Chapter 07 -

Linked List

Nor Bahiah Hj Ahmad & Dayang Norhayati A.Jawawi
School of Computing

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Objectives

At the end of the class, students are expected to :

- Describe linear list concepts using array and linked list.
- Able to lists variations of linked list and basic operations of linked lists.
- Able to explain in detail the implementation and operations of link lists using pointers.
- Able to write program that can implement linked list concept.

Introduction

Lists Definition

- Lists is a group of objects which is organized in sequence.
- List categories: linear list and nonlinear list.
- Linear list – a list in which the data is organized in sequence, example: array, linked list, stack and queue
- Non-Linear list – a list in which the data is stored not sequence, example: tree and graph

Introduction to Linear List

- **Array and linked** lists are linear lists that doesn't have any restrictions while implementing operations such as, insertion, deletion and accessing data in the lists.
- The operations can be done in any parts of the lists, either in the front lists, in the middle or at the back of the lists.

- **Stack and queue** is a linear lists that has restrictions while implementing its operations.
  - **Stack** – to insert, delete and access data can only be done at the top of the lists.
  - **Queue** - Insert data in a queue can be done at the back of the lists while to delete data from a queue can only be done at the front list.

Introduction to Linear List

Pelajar

Indeks	Nama	Kursus	Tahun
[0]	Aziz Nabil	Sains Komputer	2
[1]	Boh Guan	Sains Komputer	1
[2]	Durrani Nukman	Kej. Elektrik	3
[3]	Mohd Saufi	Sains Pendidikan	2
[4]	Nafisah Nordin	Sains Komputer	1
[5]	Safinatun Najah	Pengurusan Komputer	3
[6]			
[7]			

- An array named Pelajar which contains attributes nama pelajar, kursus and tahun Pelajar.
- The array is sorted and can only be accessed based on the index or subscript of the array.
- Example: to access information for a student named Mohd Saufi, we can use:

  Pelajar[3].Nama, Pelajar[3].Kursus dan Pelajar[3].Tahun.

Introduction to Linear List

- Linked lists which contain several nodes which is sorted in ascendeng order.
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- Need pointer variable *Head [Senarai]*: to point to the first node

Introduction to Linear List

- Basic operations for linear lists:
  - Insert new data in the lists.
  - Delete data from a lists.
  - Update data in the list.
  - Sort data in the lists and
  - Find data in the list.

Array as linear list

- Sized is fixed during array declaration.
- Data insertion is limited to array size
- In order to insert data, need to check whether the array is full or not. If the array is full, the insertion cannot be done.

Array as linear list

- Data in the array can be accessed at random using the index of the array.
- Example, if we access Nama, Kursus and Tahun for Pelajar[3] information of Pelajar[3], Mohd Saufi taking Sains Pendidikan course and in year 2 will be given.
- By random access, accessing data in an array can be done faster.

Pelajar

Indeks	Nama	Kursus	Tahun
[0]	Aziz Nabil	Sains Komputer	2
[1]	Boh Guan	Sains Komputer	1
[2]	Durrani Nukman	Kej. Elektrik	3
[3]	Mohd Saufi	Sains Pendidikan	2
[4]	Nafisah Nordin	Sains Komputer	1
[5]	Safinatun Najah	Pengurusan Komputer	3
[6]			
[7]			

Array Implementation … the drawbacks

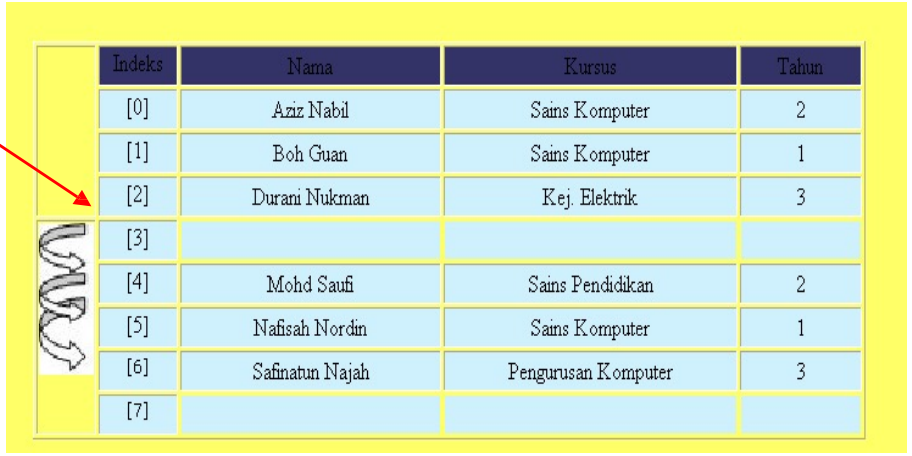
- Requires an estimate of the maximum size of the list
  - waste space
- printList and find: linear access
- findKth: constant
- insert and delete: slow
  - insert at position 0 (making a new element)
    - requires first pushing the entire array down one spot to make room
  - delete at position 0
    - requires shifting all the elements in the list up one
  - On average, half of the lists needs to be moved for either operation

Array Implementation … the drawbacks

- Need space to insert item in the middle of the list.
- Insert Fatimah Adam in between students named Durrani Nukman and Mohd Saufi.
- insert at index 3 : requires first pushing the entire array from index 3 down one spot to make room

Pelajar

Indeks	Nama	Kursus	Tahun
[0]	Aziz Nabil	Sains Komputer	2
[1]	Boh Guan	Sains Komputer	1
[2]	Durrani Nukman	Kej. Elektrik	3
[3]	Mohd Saufi	Sains Pendidikan	2
[4]	Nafisah Nordin	Sains Komputer	1
[5]	Safinatun Najah	Pengurusan Komputer	3
[6]			
[7]			



UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Array Implementation … the drawbacks

Indeks	Nama	Kursus	Tahun
[0]	Aziz Nabil	Sains Komputer	2
[1]	Boh Guan	Sains Komputer	1
[2]	Durani Nukman	Kej. Elektrik	3
[3]	Fatimah Adam	Kej. Awam	2
[4]	Mohd Saufi	Sains Pendidikan	2
[5]	Nafisah Nordin	Sains Komputer	1
[6]	Safinatun Najah	Pengurusan Kompuer	3
[7]			

New item is inserted at index 3, after shifting the data from index 3 onwards. .

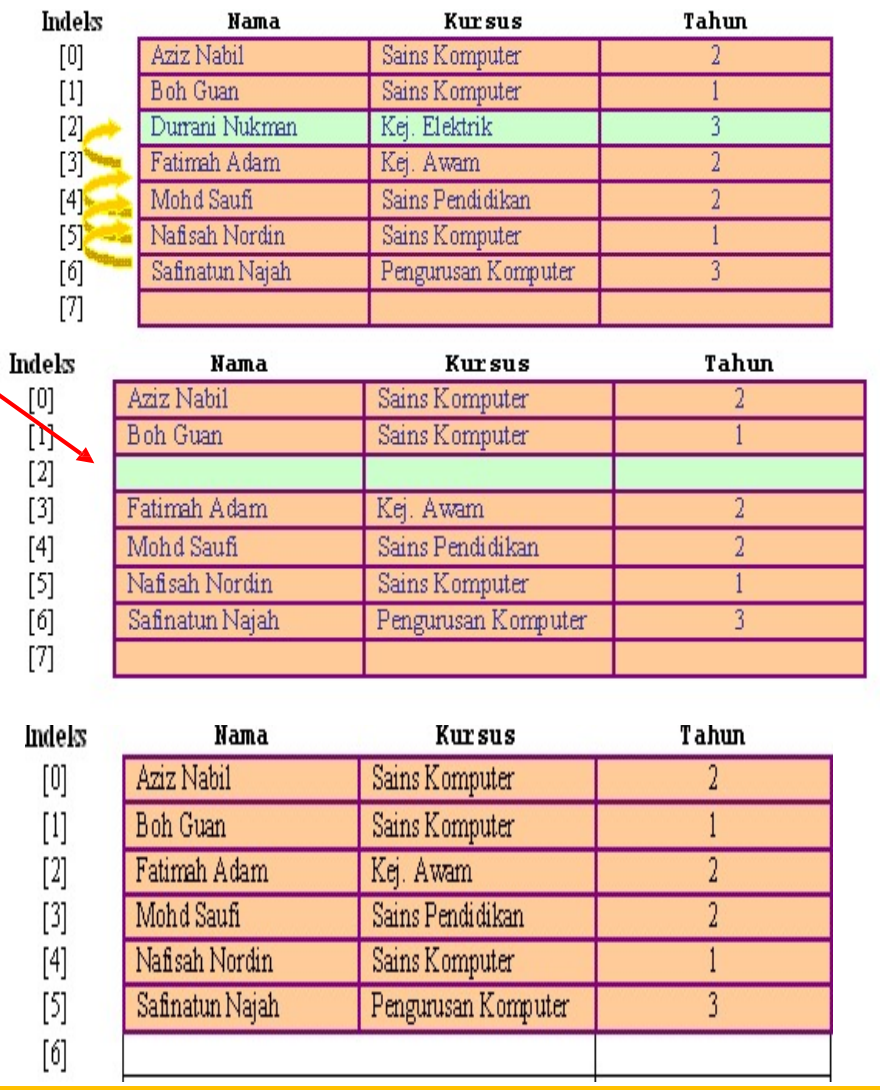
UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Array Implementation … the drawbacks

To delete item in the middle of the array will leave a blank space in the middle.

It requires shifting all the elements in the list up one in order to eliminate the space..

Example: when information about Durrani Nukman is deleted, all elements below it, is shifted up.



Indeks	Nama	Kursus	Tahun
[0]	Aziz Nabil	Sains Komputer	2
[1]	Boh Guan	Sains Komputer	1
[2]	Durrani Nukman	Kej. Elektrik	3
[3]	Fatimah Adam	Kej. Awam	2
[4]	Mohd Saufi	Sains Pendidikan	2
[5]	Nafisah Nordin	Sains Komputer	1
[6]	Safinatun Najah	Pengurusan Komputer	3
[7]			

Indeks	Nama	Kursus	Tahun
[0]	Aziz Nabil	Sains Komputer	2
[1]	Boh Guan	Sains Komputer	1
[2]			
[3]	Fatimah Adam	Kej. Awam	2
[4]	Mohd Saufi	Sains Pendidikan	2
[5]	Nafisah Nordin	Sains Komputer	1
[6]	Safinatun Najah	Pengurusan Komputer	3
[7]			

Indeks	Nama	Kursus	Tahun
[0]	Aziz Nabil	Sains Komputer	2
[1]	Boh Guan	Sains Komputer	1
[2]	Fatimah Adam	Kej. Awam	2
[3]	Mohd Saufi	Sains Pendidikan	2
[4]	Nafisah Nordin	Sains Komputer	1
[5]	Safinatun Najah	Pengurusan Komputer	3
[6]			

Pointer Implementation (Linked List)

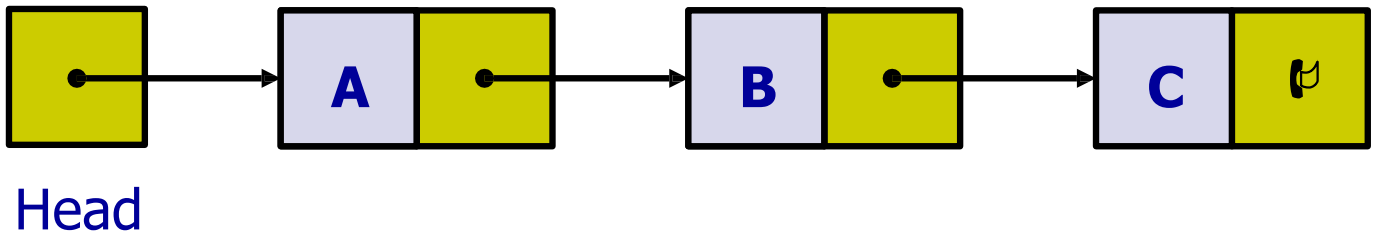
- Ensure that the list is not stored contiguously
  - use a linked list
  - a series of structures that are not necessarily adjacent in memory
- Each node contains the element and a pointer to a structure containing its successor
  - the last cell's next link points to NULL
- Compared to the array implementation,
  - the pointer implementation uses only as much space as is needed for the elements currently on the list
  - but requires space for the pointers in each cell

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

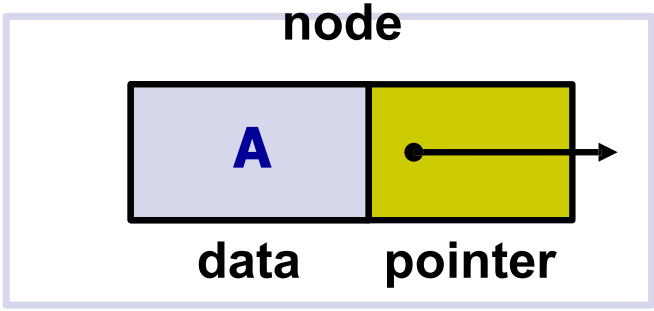
Linked list variations

- Singly linked list
- Doubly linked list
- Circular linked list
- Circular doubly linked list
- Sorted linked list
- Unsorted linked list

Singly Linked Lists



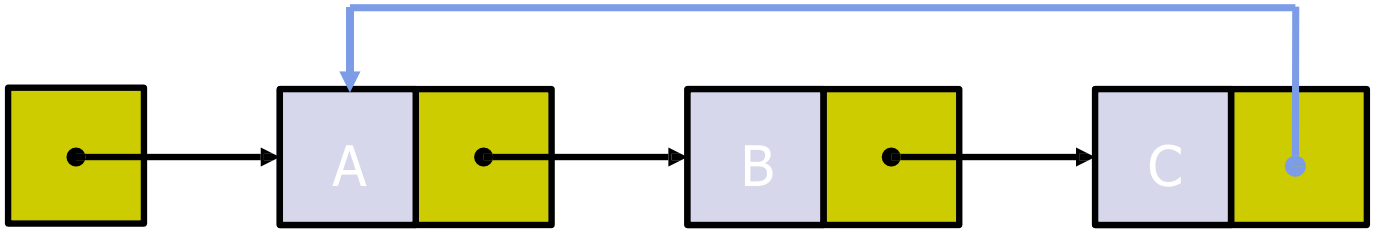
- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- *Head*: pointer to the first node
- The last node points to `NULL`



UTM UNIVERSITI TEKNOLOGI MALAYSIA

Variations of Linked Lists

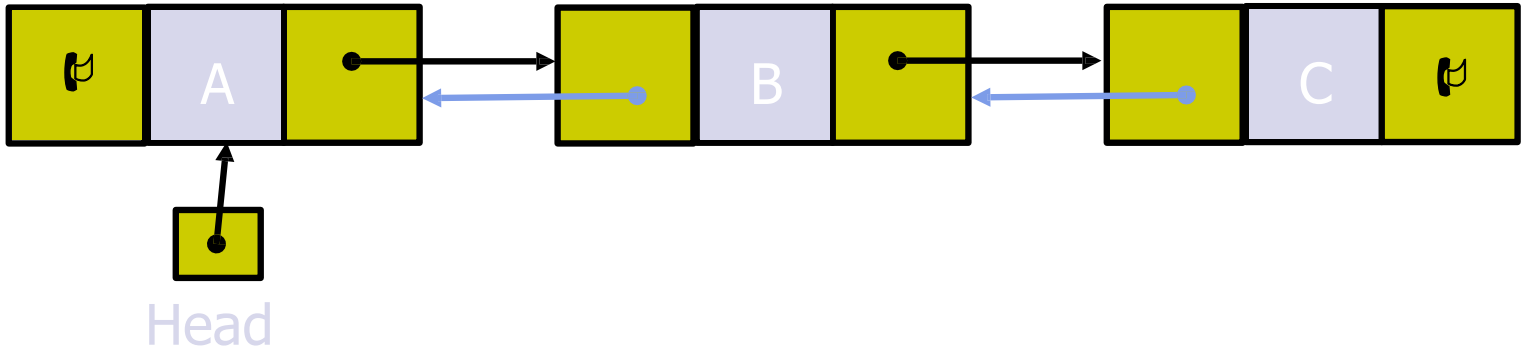
- *Circular linked lists*
  - The last node points to the first node of the list



- How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)

Variations of Linked Lists

- *Doubly linked lists*
  - Each node points to not only successor but the predecessor
  - There are two `NULL:` at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards

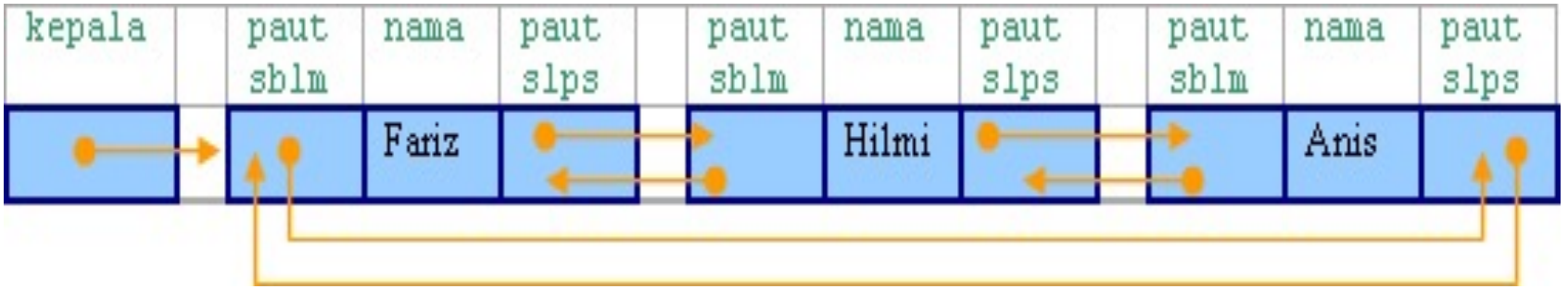


UTM UNIVERSITI TEKNOLOGI MALAYSIA

Variations of Linked Lists

Circular doubly linked list

- No `NULL` value at the first and last nodes in the list
- Convenient to traverse lists backwards and forwards

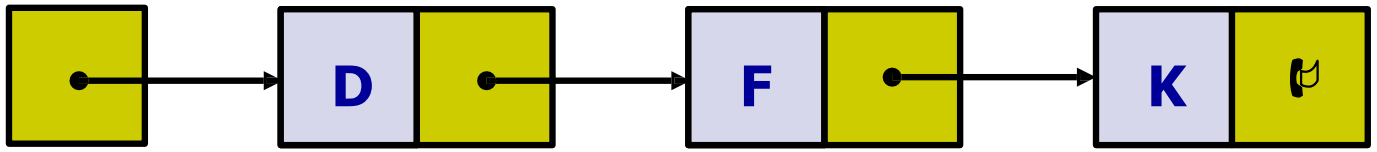


UTM UNIVERSITI TEKNOLOGI MALAYSIA

Variations of Linked Lists

Sorted Linked list :

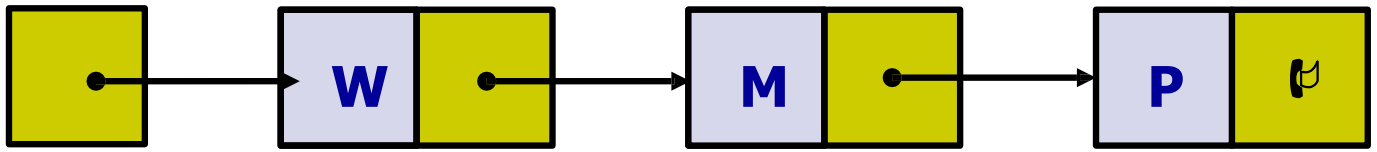
The nodes in the lists is sorted in certain order.



Head

UnSorted Linked list :

The nodes in the lists is not sorted in any order.



Head

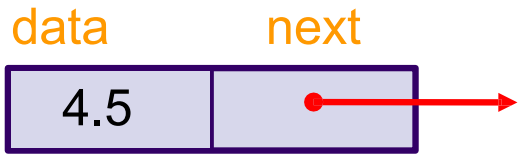
LINK LIST IMPLEMENTATION

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Linked List Implementation

- We need two classes: **Node** and **List**
- Declare `Node` class for the nodes which contains `data` and `next`: a pointer to the next node in the list

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

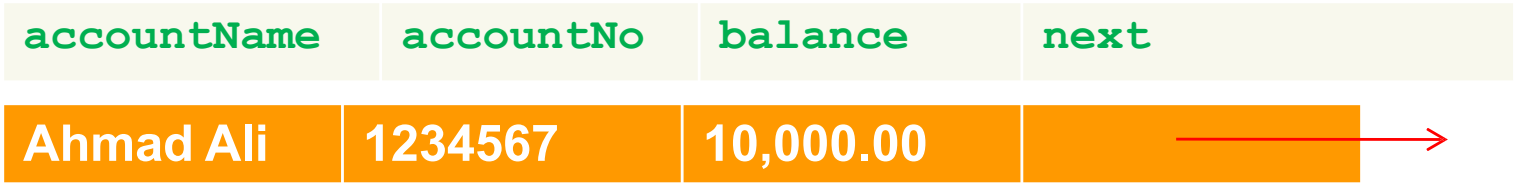


UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Example :
Declaring a node for class account

Create a node for class account using struct

```
struct nodeAccount {
        char accountName[20];
        char accountNo[15];
        float balance;
        nodeAccount *next;
};
```



UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Exercise – Declaring a node

bookTitle	ISBNno	price	next
Link List Guide	1234-5678	40.00	NULL

Figure above represents a node in a link list that store information for book.
The node consists of the following attributes :

bookTitle, ISBNno, price and pointer next.

Write a class/struct declaration for the node that is able to store the information.

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`
  - Operations on `List`

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

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

UTM UNIVERSITI TEKNOLOGI MALAYSIA

A Simple Linked List Class

Operations of `List`

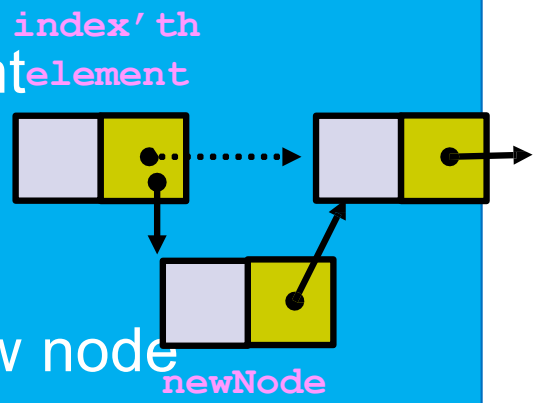
- **`IsEmpty`**: determine whether or not the list is empty
- **`InsertNode`**: insert a new node at a particular position
- **`FindNode`**: find a node with a given value
- **`DeleteNode`**: delete a node with a given value
- **`DisplayList`**: print all the nodes in the list

UTM UNIVERSITI TEKNOLOGI MALAYSIA

Inserting a new node

X

- `Node* InsertNode(double x)`
  - Insert a node with data equal to `x`. After insertion, this function generates a sorted list : in ascending order.
  - Find the location of the value to be inserted so that the value will be in the correct order in the list. **(i.e., when `index = 0`, insert the node as the first element; when `index = 1`, insert the node after the first element, and so on)**
- Steps
  1. Locate `position to insert` the element
  2. Allocate memory for the new node
  3. Point the new node to its successor
  4. Point the new node's predecessor to the new node



UTM UNIVERSITI TEKNOLOGI MALAYSIA

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)

Inserting a new node (will create a sorted list)

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

Try to locate the index of the node being inserted.

Inserting a new node

```
Node* List::InsertNode(double x) {

        int  currIndex  =       0;
        Node* currNode =        head;
        Node* prevNode =        NULL;
        while (currNode && x > currNode->data) {
                prevNode = currNode;
                currNode = currNode->next;
                currIndex++;
        }

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

Create a new node

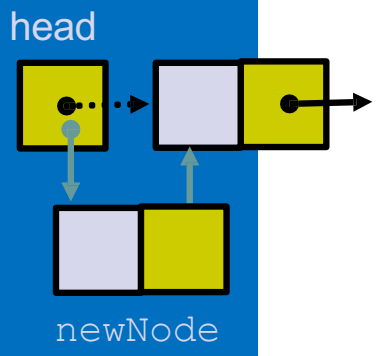
Inserting a new node

```
Node* List::InsertNode(double x) {

        int  currIndex  =       0;
        Node* currNode =        head;
        Node* prevNode =        NULL;
        while (currNode && x > currNode->data) {
                prevNode = currNode;
                currNode = currNode->next;
                currIndex++;
        }

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

Insert as first element

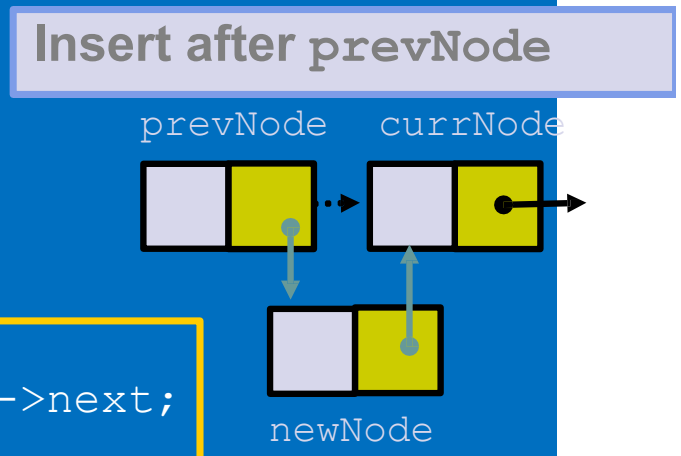


Inserting a new node

```
Node* List::InsertNode(double x) {

        int  currIndex  =       0;
        Node* currNode =        head;
        Node* prevNode =        NULL;
        while (currNode && x > currNode->data) {
                prevNode = currNode;
                currNode = currNode->next;
                currIndex++;
        }

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



Inserting a new node at certain index (will create unsorted list)

```
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`.

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

Create a new node

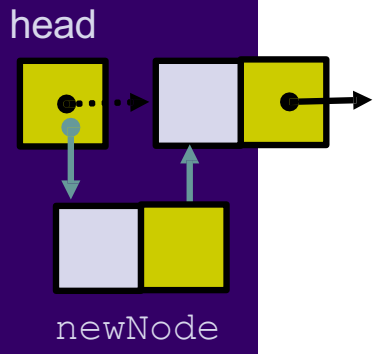
Inserting a new node

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

Insert as first element



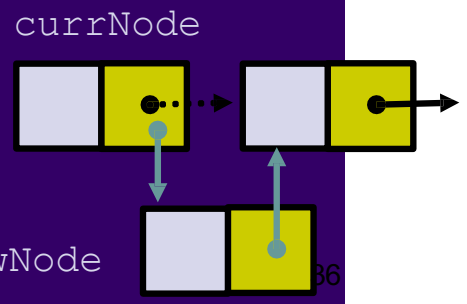
Inserting a new node

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

Insert after currNode



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;
        else
                return 0;
}
```

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

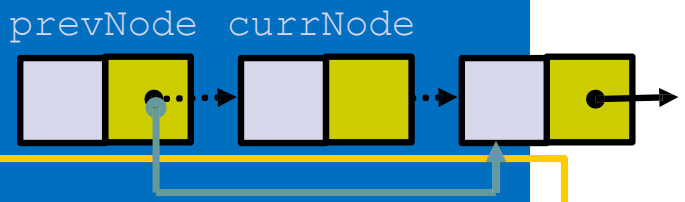
Deleting a node

```
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;
        }
        return 0;
```

Try to find the node with its value equal to `x`

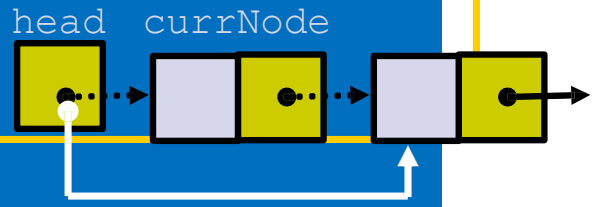
Deleting a node

```
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;
        }
        return 0;
}
```



Deleting a node

```
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;
        }
        return 0;
}
```



UTM UNIVERSITI TEKNOLOGI MALAYSIA

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;
}
```

UTM UNIVERSITI TEKNOLOGI MALAYSIA

Destroying the list : `~List()`

- Use the destructor to release all the memory used by the list.
- Step through the list and delete each node

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

4

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Using List

```
int main(void)
{
        List list;
        list.InsertNode(7.0); // successful
        list.InsertNode(5.0); // successful
        list.InsertNode(6.0); // successful
        list.InsertNode(4.0); // successful
        // print all the elements
        list.DisplayList();
        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;
}
```

result

```
4
5
6
7
Number of nodes in the list: 4
5.0 found
4.5 not found
4
5
6
Number of nodes in the list: 3
```

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
  - Dynamic: a linked list can easily grow and shrink in size.
    - We don't need to know how many nodes will be in the list. They are created in memory as needed.
    - In contrast, the size of a C++ array is fixed at compilation time.

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Easy and fast insertions and deletions

- To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
- With a linked list, no need to move other nodes. Only need to reset some pointers.

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Linked List - Conclusion

- Try make your own conclusion by comparing array list and linked list in term of:
  - Size of items/nodes in the list
  - Add new items/nodes at the beginning or in the middle of the list
  - Delete items at the beginning or in the middle of the list
  - Get items at nodes *n*

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Lab Exercise

- Write a complete link list program based on the implementation of the following classes and functions given in the notes.
  - Class **`Node`** and **`List`**
  - **`Constructor`** and **`Destructor`**
  - **`IsEmpty()`**,**`InsertNode()`**, **`FindNode()`** **`DeleteNode()`** and **`DisplayList()`**
  - Provide main() program to execute the list.

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Thank You

innovative ● entrepreneurial ● global