MODULE 7

LINKED LIST

DATA STRUCTURE AND ALGORITHMS

FACULTY OF COMPUTING
UNIVERSITI TEKNOLOGI MALAYSIA

OBJECTIVES FOR STUDENTS

- To understand the linked list concept.
- To work with pointers in linked list.
- To implement linked list operations in problem solving.

KEY CONCEPT

1.0 INTRODUCTION TO LINEAR LIST

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

1.2 **Array** and **linked lists** are linear lists that do not 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 of the lists, in the middle or at the back of the lists.

1.3 **Stack** and **queue** are two types of linear lists that have restrictions while implementing their 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 of the list.

1.4 Example of array as a list is shown in Figure 7.1, whereby, an array of objects named **Student** which contains attributes such as **name**, **course** and **year**. The element of the array can only be accessed based on the index or subscript of the array.
- In order to access all information for a student named **Mohd Saufi**, located at index 3 of the array, we can access the element using the subscript as follows :

cout << Student[3].name << Student[3].course << Student[3].year

Student

index	name	course	year
[0]	Abu Umar	Engineering	1
[1]	Tan Ai Tee	Education	2
[2]	Durrani Nukman	Physic	1
[3]	Mohd Saufi	Mathematics	1
[4]	Nur Ilahi	Islamic Study	2
[5]	Ahmad Ali	Computer Science	3
[6]			
[7]			

Figure 7.1 **Student** Array

1.5 Figure 7.2 show a **linked lists** which contains several nodes which are sorted in ascending order. Each node contains at least:

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

Linked list need a pointer variable, named **head** to point to the first node.

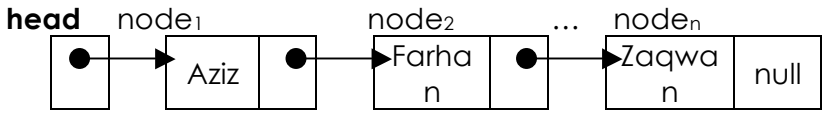


Figure 7.2 Sorted linked list

1.6 Basic **operations** for linear lists:

- Insert new data in the lists.
- Delete data from the lists.
- Update data in the list.
- Sort data in the lists and
- Find data in the list.

2.0 ARRAY AS A LINEAR LIST

2.1 Size of array is fixed during array declaration.

- **Disadvantages of array** - data insertion is limited to the size. In order to insert data, we need to check whether the array is full or not. If the array is full, the insertion cannot be done.
- **Advantage of array** - data in the array can be accessed at random using the index of the array. For example, in Figure 7.1, when we execute the statement:

  cout << Student[3].name << Student[3].course << Student[3].year

  the information of **Student[3]**, Mohd Saufi taking Mathematics course and in year 1 will be the output. Accessing any data by using random access in an array can be done faster compare with accessing the data sequentially from the array.

2.2 The drawbacks of array implementation:

- Requires an estimate of the maximum size of the list. May waste space, if the memory is not fully utilized.
- Linear access to print the whole content of the list or to find an element from the list will take longer time, O(n).
- Insert and delete element are slow.
  - To insert element at index 0 which already occupied by other element, requires first pushing the entire array down one spot to make room
  - To delete at index 0 - requires shifting all the elements in the list up.
  - On average, half of the lists need to be moved for either operation.
- Need space to insert item in the middle of the list.
- Example – To insert **Fatimah Adam** in between students named **Durrani Nukman** and **Mohd Saufi** as shown in Figure 7.3, it requires first pushing the entire array from index 3 down one spot to make room, as shown in Figure 7.4

Student

index	name	course	year
[0]	Abu Umar	Engineering	1
[1]	Tan Ai Tee	Education	2
[2]	Durrani Nukman	Physic	1
[3]	Mohd Saufi	Mathematics	1
[4]	Nur Ilahi	Islamic Study	2
[5]	Ahmad Ali	Computer Science	3
[6]			
[7]			

Fatimah Adam →

Figure 7.3 Insert element between Durani Nukman and Mohd Saufi

Student

index	name	course	year
[0]	Abu Umar	Engineering	1
[1]	Tan Ai Tee	Education	2
[2]	Durrani Nukman	Physic	1
[3]			
[4]	Mohd Saufi	Mathematics	1
[5]	Nur Ilahi	Islamic Study	2
[6]	Ahmad Ali	Computer Science	3
[7]			

Figure 7.4 Push the entire array from index 3 down one spot

- Insert **Fatimah Adam** at empty space at index 3. Shown in Figure 7.5.
- New item is inserted at index 3, after shifting the data from index 3 onwards.

Student

index	name	course	year
[0]	Abu Umar	Engineering	1
[1]	Tan Ai Tee	Education	2
[2]	Durrani Nukman	Physic	1
[3]	**Fatimah Adam**	**Civil Engineering**	**2**
[4]	Mohd Saufi	Mathematics	1
[5]	Nur Ilahi	Islamic Study	2
[6]	Ahmad Ali	Computer Science	3
[7]			

Figure 7.5 Insert **Fatimah Adam** at index 3

- To **delete item** in the middle of the array will leave a blank space in the middle. Example, there is empty space after delete **Durrani Nukman** at index 2 as shown in Figure 7.6.

Student

index	name	course	year
[0]	Abu Umar	Engineering	1
[1]	Tan Ai Tee	Education	2
[2]			
[3]	Fatimah Adam	Civil Engineering	2
[4]	Mohd Saufi	Mathematics	1
[5]	Nur Ilahi	Islamic Study	2
[6]	Ahmad Ali	Computer Science	3
[7]			

Figure 7.6 Delete **Durrani Nukman** at index 2

- It requires shifting all the elements in the list one position up in order to eliminate the space. For example: when information about **Durrani Nukman** is deleted, all elements under it are shifted up. Shown in Figure 7.7.

Student

index	name	course	year
[0]	Abu Umar	Engineering	1
[1]	Tan Ai Tee	Education	2
[2]	Fatimah Adam	Civil Engineering	2
[3]	Mohd Saufi	Mathematics	1
[4]	Nur Ilahi	Islamic Study	2
[5]	Ahmad Ali	Computer Science	3
[6]			
[7]			

Figure 7.7 Shift up all elements below **Durrani Nukman**

3.0 LINKED LIST

3.1 **Pointer** Implementation (Linked List)

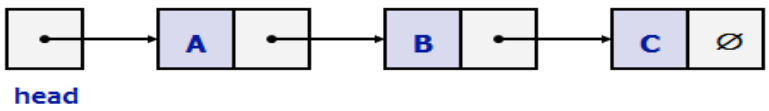
- Ensure that the list is not stored contiguously
- Use a linked list - a series of structures that is not necessarily adjacent in memory

- 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 needed for the elements currently on the list
  - but requires space for the pointers in each cell

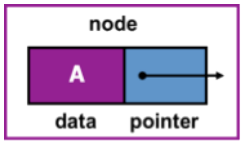
3.2 Variations of **linked lists**

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

3.3 **Singly** Linked Lists

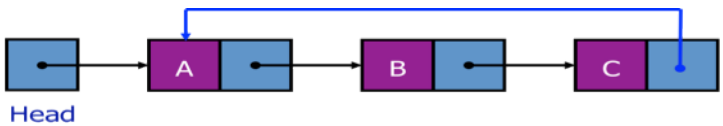


- A **linked list** is a series of connected nodes. Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- **Head**: pointer to the first node
- The last node points to **NULL**



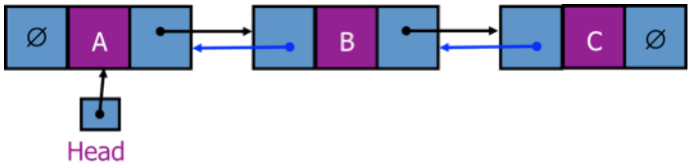
3.4 **Circular** linked lists

- The last node points to the first node of the list



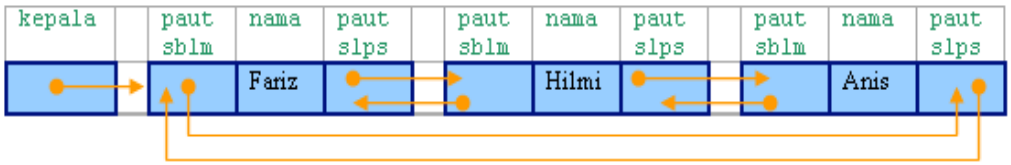
3.5 **Doubly** linked lists

- 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



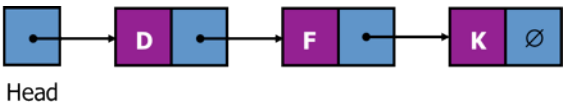
3.6 **Circular doubly** linked list

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



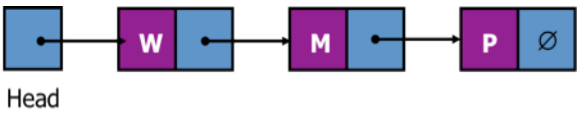
3.7 **Sorted** Linked list

- The nodes in the list are sorted in a certain order.



3.8 **Unsorted** Linked list

- The nodes in the lists are not sorted in any order.



4.0 IMPLEMENTATION OF LINKED LIST

4.1 A Simple Linked List **Class**

- We need two classes declaration: **Node** and **List**
- Declaration of **Node** class for the nodes require at least two attributes:
  - **data**: double-type data in this example
  - **next**: a pointer to the next node in the list

```
1  // Program 7.1a
2  class Node {
```

```
3  public:
4          double data; // data
5          Node* next;    // pointer to next
6  };
```

- Declaration of **List** class for the linked list contain at list
  - **head**: a pointer to the first node in the list.
  - Since the list is empty initially, **head** is set to **NULL**

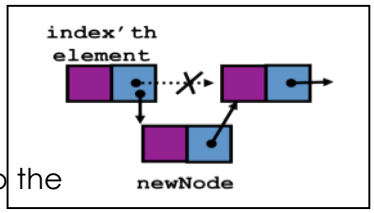
```
1  // Program 7.1b
2  class List {
3  public:
4          List(void) { head = NULL; }    // constructor
5          ~List(void);                   // destructor
6          bool IsEmpty() { return head == NULL; }
7          Node* InsertNode(double x);
8          int FindNode(double x);
9          int DeleteNode(double x);
10         void DisplayList(void);
11 private:
12         Node* head;
13 };
```

4.2 **List** Operations

- **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

4.3 Inserting a new node to the list

- **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 right order in the list.
  - Steps:
    i. Locate index to insert the element.
    ii. Allocate memory for the new node.
    iii. Point the new node to its successor.
    iv. Point the new node's predecessor to the new node.



4.4 Possible cases of **InsertNode**

i. Insert into an empty list
ii. Insert in front
iii. Insert at back
iv. Insert in middle

4.5 But, in fact, we 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)

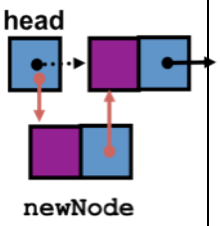
4.6 **InsertNode** Source Codes

```
1  // Program 7.2
2  Node* List::InsertNode(double x) {
3
4     Node* currNode =      head;
5     Node* prevNode        =      NULL;
6     while (currNode && x > currNode->data)
7     {    prevNode = currNode;
8       currNode = currNode->next;
9     }
10
11
12    Node* newNode=        new    Node;
13    newNode->data =       x;
14
15    if (prevNode == NULL)
16    {
17        newNode->next     = head;
18        head              = newNode;
19    }
20
21    else
22    {
23        newNode->next = prevNode->next;
24        prevNode->next = newNode;
25    }
26
27    return newNode;
28 }
```

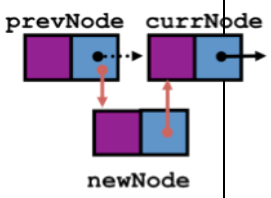
Locate the position of the node

Create a new

Insert as first element



Insert after `prevNode`



4.7 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.

```
1  // Program 7.3
2  int List::FindNode(double x) {
```

```
3       Node* currNode    =    head;
4       int currIndex  =    1;
5       while (currNode && currNode->data != x) {
6               currNode    = currNode->next;
7               currIndex++;
8       }
9       if (currNode)
10              return currIndex;
11      else
12              return 0;
13  }
```

4.8 Deleting a node

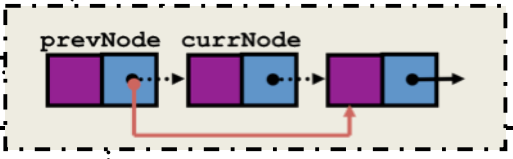
- **int DeleteNode(double x)**
  - Delete a node with the value equal to **x** from the list.
  - If such a node is found, return its position. Otherwise, return 0.
- Steps
  - Find the desirable node (similar to **FindNode**)
  - Release the memory occupied by the found node
  - Set the pointer of the 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

4.9 Delete node source codes

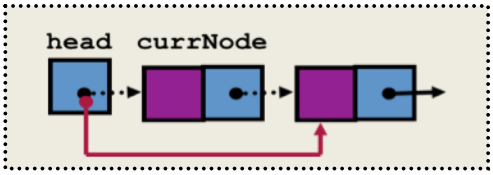
```
1   // Program 7.4
2   int List::DeleteNode(double x) {
3     Node* prevNode = NULL;
4     Node* currNode = head;
5     int currIndex = 1;
6     while (currNode && currNode->data != x)
7     {
8         prevNode = currNode;
9         currNode = currNode->next;
10        currIndex++;
11    }
12
13    if (currNode) {
14        if (prevNode) {
15              prevNode->next = currNode->next;
16              delete currNode;
17        }
18        else {
```

Find the node with its value equal to x

Delete the node in the middle or back list



```
19                  head = currNode->next;
20                  delete currNode;
21              }
22              return currIndex;
23          }
            return 0;
        }
```



Delete the node in the first list

4.10 Printing all the elements in the list

- **void DisplayList(void)**
  - Print all the data of all elements in the list
  - Print the number of the nodes in the list

4.11 Print data in the list source codes

```
1   // Program 7.5
2   void List::DisplayList()
3   {
4     int num          =      0;
5     Node* currNode   =      head;
6     while (currNode != NULL){
7           cout << currNode->data << endl;
8           currNode     =      currNode->next;
9           num++;
10    }
11    cout << "Number of nodes in the list: " << num << endl;
12  }
13
```

4.12 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.

4.13 Destroying the list source codes (destructor)

- All the nodes in the linked list will be destroyed one by one, starting from the first node until the last node

```
1   // Program 7.6
2   List::~List(void) {
3     Node* currNode = head, *nextNode = NULL;
4     while (currNode != NULL)
5     {
6           nextNode     =      currNode->next;
7           // destroy the current node
```

```
8           delete currNode;
9           currNode    =    nextNode;
10        }
11  }
```

4.14 Implementing **List**

```
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:
```

```
1   // Program 7.7
2   int main(void)
3   {
4           List list;
5           list.InsertNode(7.0);
6           list.InsertNode(5.0);
7           list.InsertNode(6.0);
8           list.InsertNode(4.0);
9           // print all the elements
10          list.DisplayList();
11          if(list.FindNode(5.0) > 0)
12                  cout << "5.0 found" << endl;
13          else
14                  cout << "5.0 not found" << endl;
15          if(list.FindNode(4.5) > 0)
16                  cout << "4.5 found" << endl;
17          else
18                  cout << "4.5 not found" << endl;
19          list.DeleteNode(7.0);
20          list.DisplayList();
21          return 0;
22  }
```

4.15 Array versus Linked Lists

- Linked lists are more complex to code and to manage compare to 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.

- 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.