SCJ2013 Data Structure & Algorithms

Linked List Implementation

Nor Bahiah Hj Ahmad & Dayang
Norhayati A. Jawawi

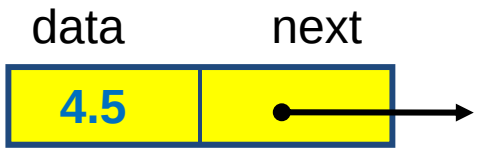
Linked List Implementation

There are 2 classes in linked list implementation:

1. Class **`Node`**
2. Classes **`list`**.

Declaration of Node

Declare **Node** class for the nodes which contains **data** and **next**, which is a pointer to the next node in the list.

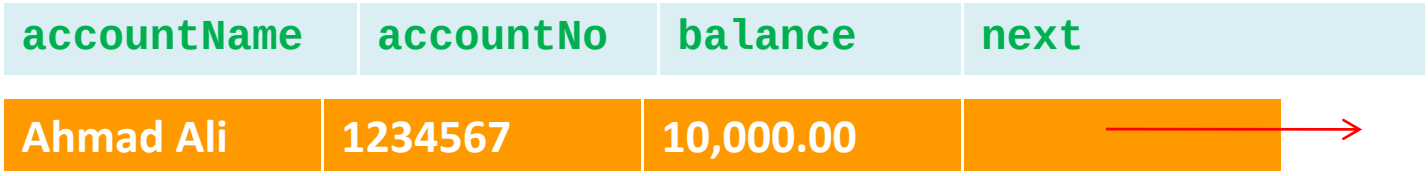



```
class Node {
public:
      double data;        // data
      Node* next;         // pointer to next node
};
```

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;
};
```



Declaration of class List

Declare class `List`, which contains

- `head`: a pointer to the first node in the list.
  The list is initially empty, `head` is set to `NULL`
- `length` : number of nodes in the list
- Operations on `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

Declaration of class List

```
class List {
public:
        // constructor
        List(void) { head = NULL; length =0;}
        // destructor
       ~List(void);

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

Insert a New Node to the List

Possible cases of `InsertNode`

1. Insert into an empty list
2. Insert in front

} case 1

3. Insert at back
4. Insert in middle

} case 2

Steps to insert a new node

1. Find the location 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

Insert a New Node to the List

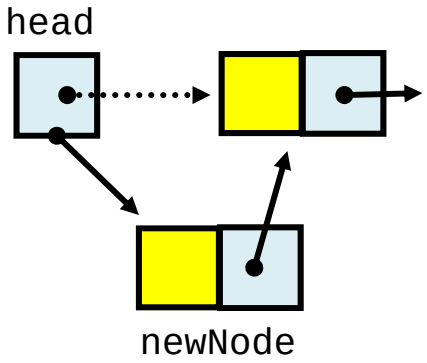
`Node* InsertNode(double x)`

- This function inserts 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.
- Allocate memory for the new node
- Insert the new node to the list.

Insert a New Node to the List

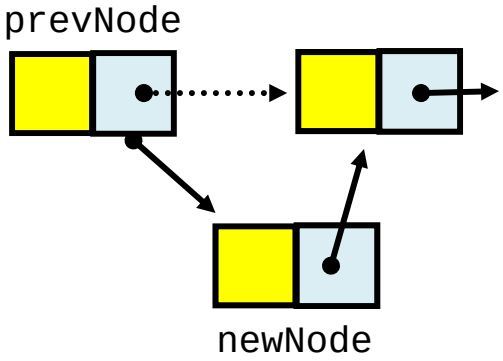
`Node* InsertNode(double x)`

- Insert at front or empty list : point head to the new node




```
newNode->next = head;
head = newNode;
```

- Insert in the middle or back list : point the new node predecessor to the new node




```
newNode->next = prevNode->next;
prevNode->next = newNode;
```

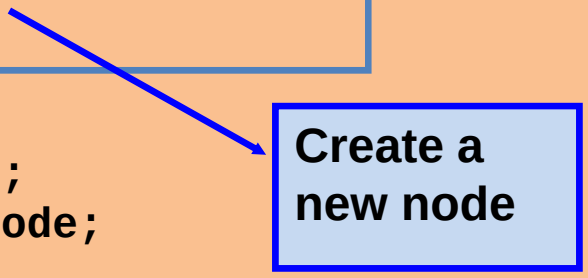
Insert Function

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

        Node* currNode =       head;
        Node* prevNode =       NULL;
        while (currNode!=NULL && x > currNode->data)
        {
                prevNode = currNode;
                currNode = currNode->next;
        }
        Node* newNode  =       new     Node;
        newNode->data  =       x;
        if (prevNode == NULL) {
                newNode->next  =       head;
                head           =       newNode;
        }
        else {
                newNode->next  =       prevNode->next;
                prevNode->next =       newNode;
        }
        length++;
         return newNode;
}
```

Locate the position of the new node being inserted.

Insert Function

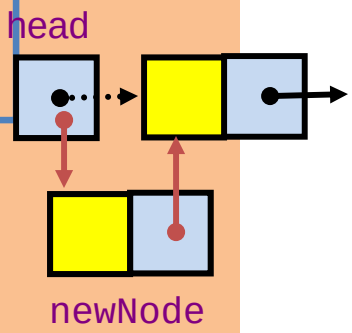
```
Node* List::InsertNode(double x) {

        Node* currNode =       head;
        Node* prevNode =       NULL;
        while (currNode!=NULL && x > currNode->data)
        {
                prevNode = currNode;
                currNode = currNode->next;
        }
        Node* newNode  =       new     Node;
        newNode->data  =       x;

         if (prevNode == Null) {
                newNode->next  =       head;
                head           =       newNode;
        }
        else {
                newNode->next  =       prevNode->next;
                prevNode->next =       newNode;
        }
         length++;
        return newNode;
}
```



Inserting a new node

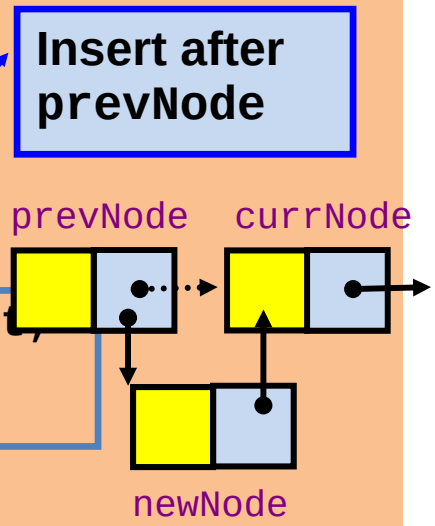
```
Node* List::InsertNode(double x) {
        Node* currNode =        head;
        Node* prevNode =        NULL;
        while (currNode!=NULL && x > currNode->data) {
                prevNode = currNode;
                currNode = currNode->next;
        }

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

Insert as first element



Inserting a new node

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

        length++;
        return newNode;
}
```



Find 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!=NULL && currNode->data != x)
       {
              currNode = currNode->next;
              currIndex++;
       }
       if (currNode!=NULL)
              return currIndex;
       else
              return 0;
}
```

Delete Node

`void DeleteNode(double x)`

- Delete a node with the value equal to x from the list.

- Steps
  - Find the node to be deleted .
  - 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.

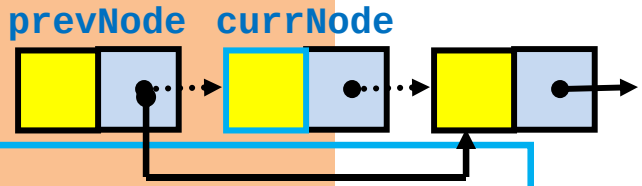
Delete Node

```
void 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;
                }
                cout << "Node " << x << "is deleted.";
                length--;
        }
         cout << "Node " << x << "is not in list.";
}
```

Find node with value equal to x

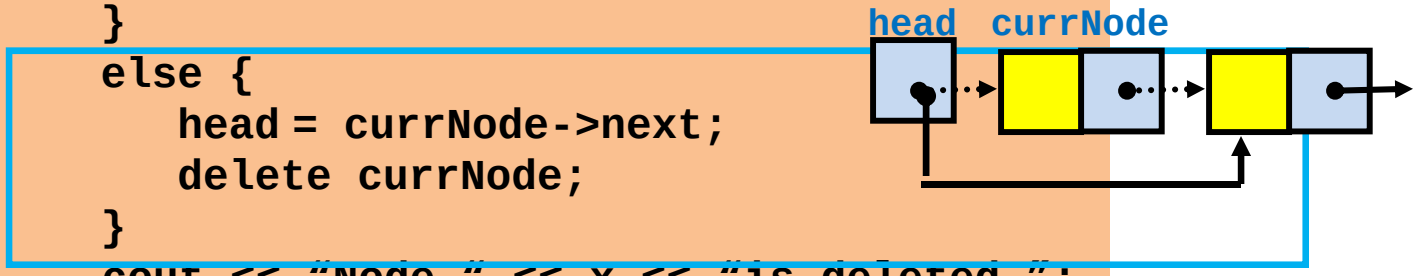
Delete Node

```
void 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;
                }
                cout << "Node " << x << "is deleted.";
                length--;
        }
         cout << "Node " << x << "is not in list.";
}
```



Delete Node

```
void 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;
                }
                cout << "Node " << x << "is deleted.";
                length--;
        }
         cout << "Node " << x << "is not in list.";
}
```



Print All Elements in the List

`void DisplayList()`

- Print the data of all the elements and
- Print the number of the nodes in the

```
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;
}
```

Destroy the List

~List()

- Destructor release all the memory used by the list.
- Delete each node in the list one by one.

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

List Implementation

```
int main()
{
   List list;
   list.InsertNode(7.0); // insert to empty list
   list.InsertNode(5.0); // insert to front list
   list.InsertNode(6.0); // insert to middle list
   list.InsertNode(14.0); // insert at the back list
   // print all the elements
   list.DisplayList();
   cout << “Insert value to be searched : “;
   cin >> value;
   if(list.FindNode(value) > 0)
        cout << value << “ is found" << endl;
   else
        cout << value << " not found" << endl;
   list.DeleteNode(7.0);
   list.DisplayList();
   return 0;
}
```

Conclusion and Summary

Implementation

- Linked lists are more complex to code and manage than arrays.

List Size

- No need to know in advanced how many nodes will be in the list. Linked list can easily grow and shrink in size dynamically.
- However, the size of a C++ array is fixed at compilation time, therefore the number of elements in the list are limited to the size.

Conclusion and Summary

Insertions and deletions

– To insert or delete an element in an array, need 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. Linked list is easier and faster to delete node in the list.

Accessing element

– In array, elements can be access at random, while in linked list item can only be accessed sequentially.

References

1. Nor Bahiah et al. *Struktur data & algoritma menggunakan C++. Penerbit UTM, 2005*
2. Richrd F. Gilberg and Behrouz A. Forouzan, "*Data Structures A Pseudocode Approach With C++*", Brooks/Cole Thomson Learning, 2001.