Data Structures in C++

CMPE226- Data Structures

Linked Lists

Adapted from Textbook (D. S. MALIK), Lecture notes (C. TURHAN) and Lecture Notes (H. QU)

Linked Lists

- Collection of components (nodes)
  - Every node (except last)
    - Contains address of the next node
- Node components
  - Data: stores relevant information
  - Link: stores address (Pointer to the next node in the list)

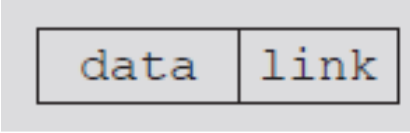


FIGURE 5-1 Structure of a node

Linked Lists (cont'd.)

- Linked List is a sequence of items arranged one after another with each item connected to the next by a link.
- Head (first)- (pointer to the first node)
  - Address of the first node in the list
- The last node points to NULL (represented by down arrow or slash)

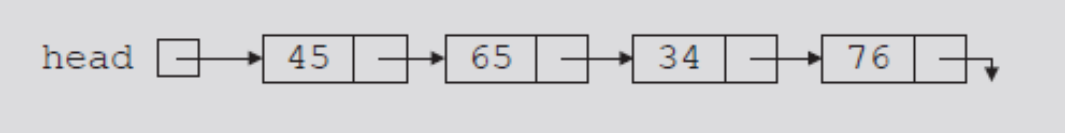


FIGURE 5-2 Linked list

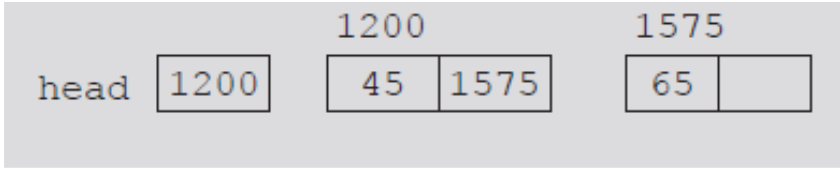
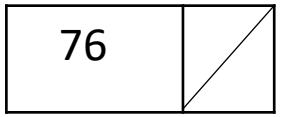


FIGURE 5-3 Linked list and values of the links



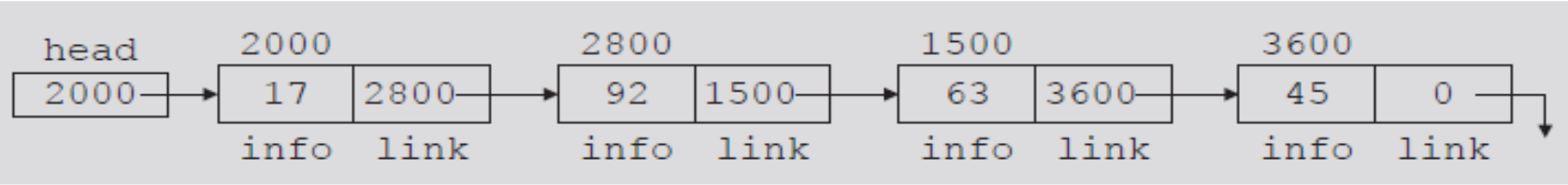
Linked Lists (cont'd.)

- Two node components: info and link.
  - Declared as a `class` or `struct`
    - Data type depends on specific application
  - Link component: pointer
    - Data type of pointer variable: node type itself

```
struct Node{
    int info;
    Node *link;
};
int main(){
    //declare head pointer of type node:
    Node *head;
}
```

Linked Lists: Some Properties

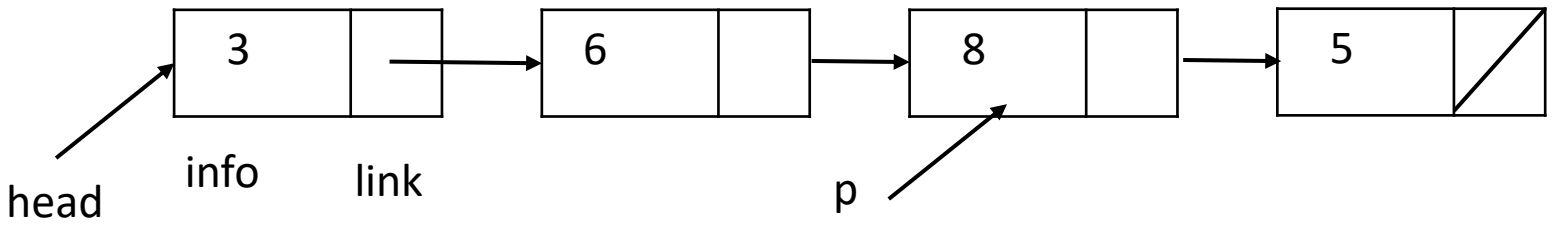
- Head stores address of first node
- Info stores information
- Link stores address of next node
- Head->info = 17
- Head->link=2800
- Head->link->info =92



	Value	Explanation
head	2000	
head->info	17	Because head is 2000 and the info of the node at location 2000 is 17
head->link	2800	
head->link->info	92	Because head->link is 2800 and the info of the node at location 2800 is 92

Linked Lists: Example

- Given the following Linked-List:

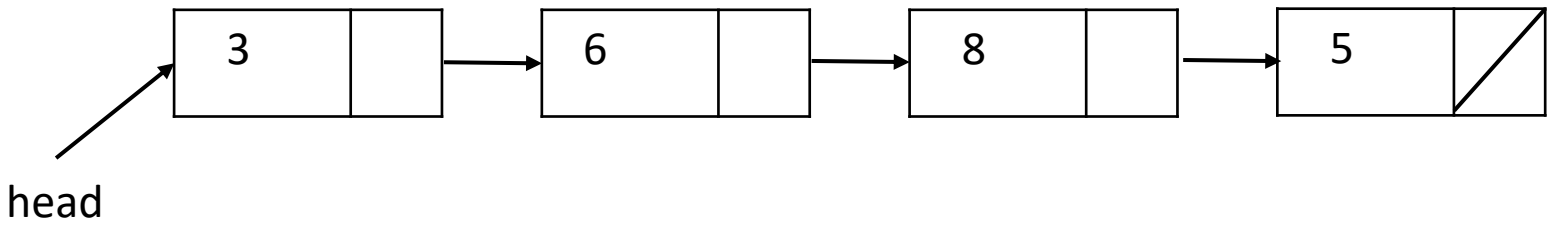


- Head stores address of first node
- Info stores information
- Link stores address of next node

Query:	Result:
head → info	
head → link → info	
p= head → link → link	
p → info	
p → link → link	
p → link → link → link	

Linked Lists: Example

- Given the following Linked-List:



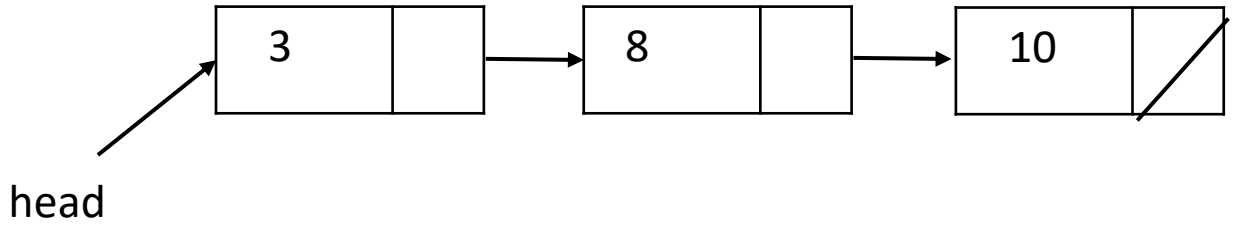
- Head stores address of first node
- Info stores information
- Link stores address of next node

Query:	**Result:**
head → info	3
head → link → info	6
p= head → link → link	p points to third node
p → info	8
p → link → link	NULL
p → link → link → link	Does not exist

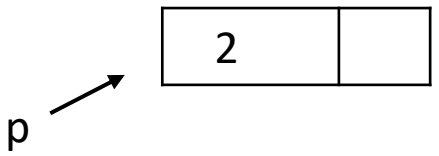
Linked Lists: Some operations

- **Inserting a new node**
  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)

Adding an element in front of a list



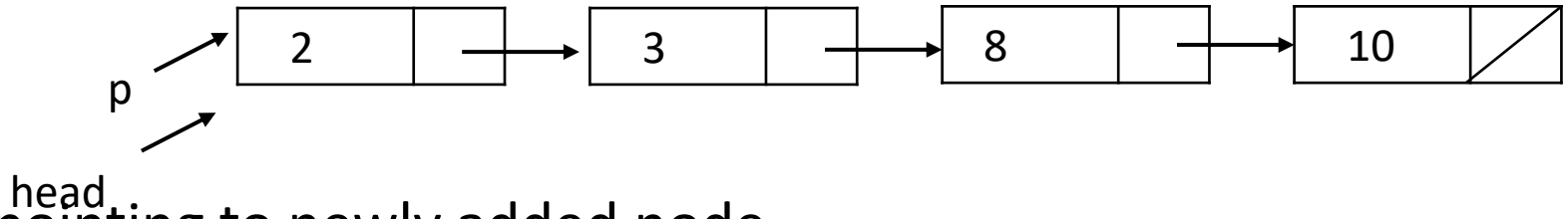
- Add integer 2 at the beginning of the list.
- **Step 1.** Obtain a new node and put 2 in it.




```
p= new node;
p->info =2;
```

- **Step 2.** next (link) should be pointing to the first node in the list (head).

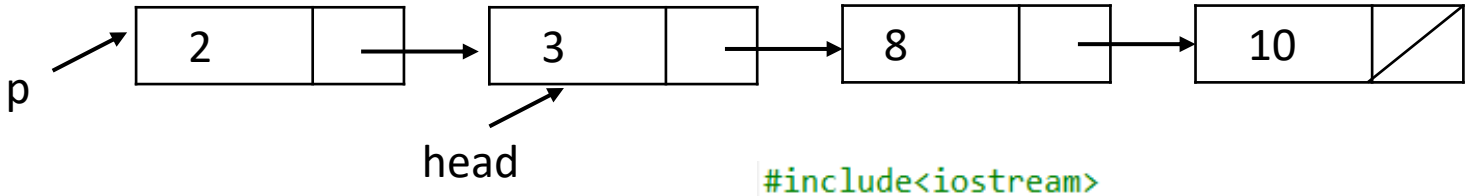
```
p->link=head;
```



- **Step 3.** New head should be pointing to newly added node.

```
head=p;
```

Adding an element in front of a list

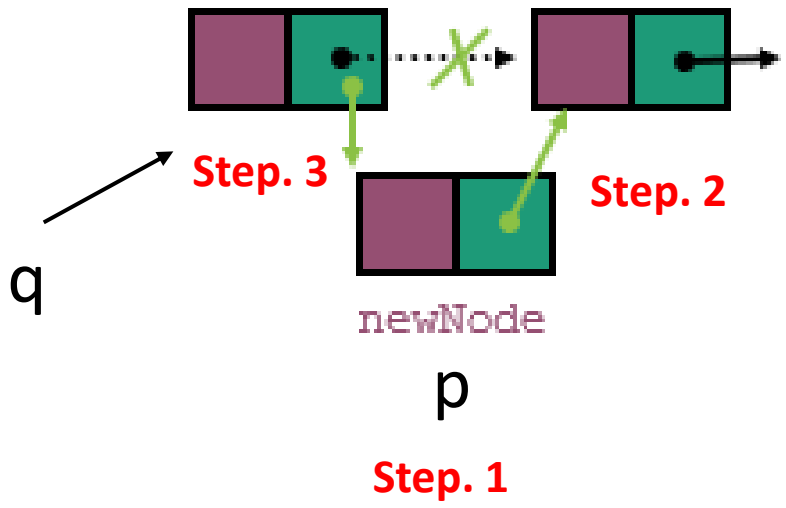



```
#include<iostream>
#include<assert.h>
using namespace std;

struct Node{
    int info;
    Node *link;
};
int main(){

    //declare head pointer of type node:
    Node *head,*p;
    int x;
    cin>>x;
    p= new Node;
    assert(p!=NULL);
    //check if list is null- terminate if node could not be created.
    //#include<assert.h>
    p->info =x;
    p->link=head;
    head=p;

}
```

Adding an element in the middle of a list

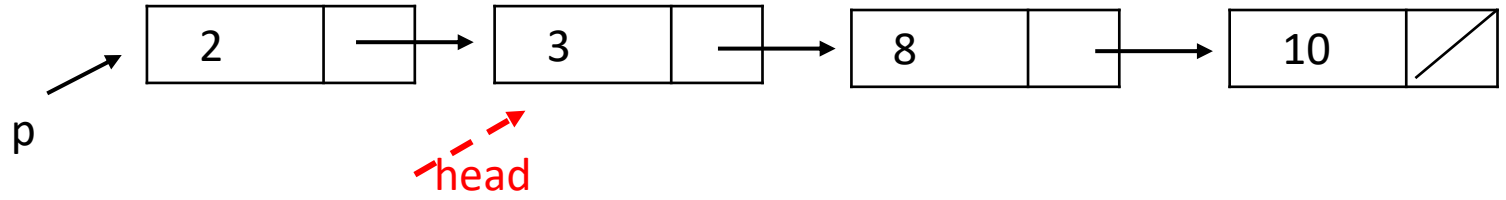
Exp. Insert a new node in the middle of a list, after the node pointed by q.

- Step1. Allocate memory for the new node
- Step2. Point the new node to its successor
- Step3. Point the new node's predecessor to the new node




```
p= new Node;
p->info =x;
p->link=q->link;
q->link=p;
```

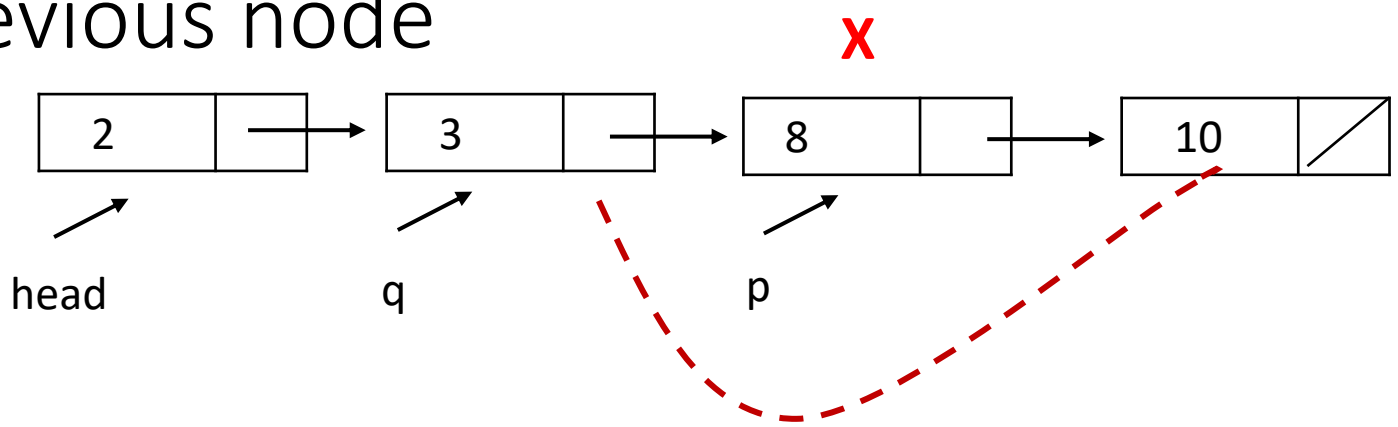
Deleting the first node of a list



- To delete node p from the list:

```
p=head;
head=p->link;
delete p; //delete the pointer to free memory allocated.
```

Deleting the node pointed by p where q points to the previous node



- To delete node p from the list:

```
p= q->link;
q->link =p->link;
delete p; //delete the pointer to free memory allocated.
```

Traversing a Linked List

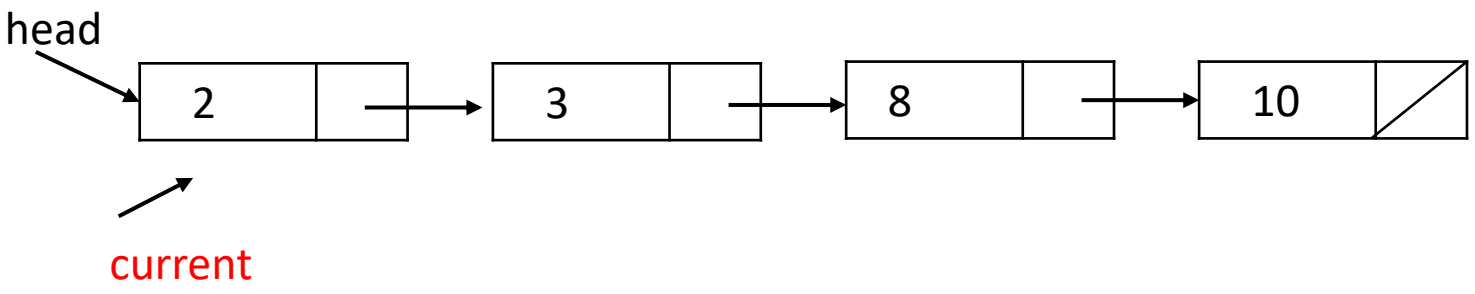
- Basic linked list operations
  - Insert item in list
  - Delete item from list
  - Search list to determine if particular item is in the list

- These operations require list traversal
  - Given pointer to list first node, we must step through list nodes

Traversing a Linked List (cont'd.)

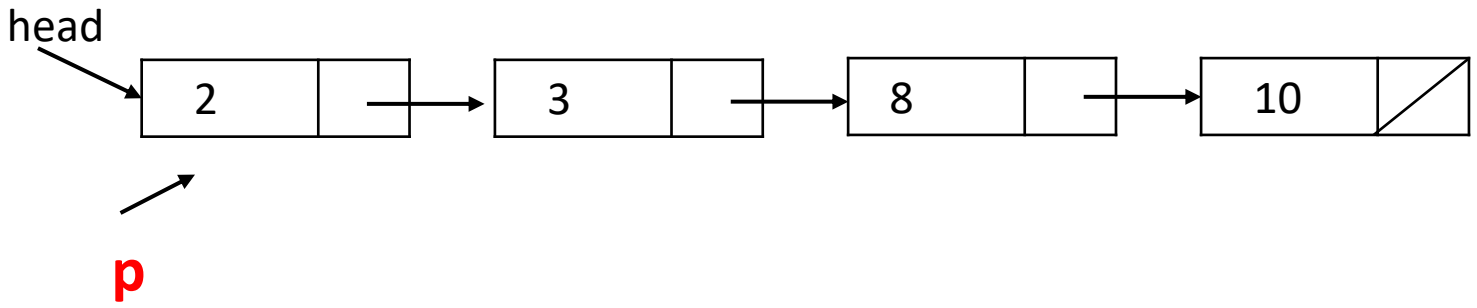
- Suppose `head` points to a linked list of numbers
  - Code outputting data stored in each node-
  - Node *head, *current;

```
current = head;

while (current != NULL)
{
    //Process current
    current = current->link;
}

current = head;

while (current != NULL)
{
    cout << current->info << " ";
    current = current->link;
}
```



Traversing the list

- E.g. Find the sum of the elements:




```
struct Node{
    int info;
    Node *link;
};
int main(){

    Node *head,*p;
    p=head;
    int sum=0;
    while(p!=NULL){
        sum=sum + p->info;
        p = p->link;
    }
```

Building a Linked List

- 2 Ways to build linked list
  - Forward
    - New node always inserted at end of the linked list
  - Backward
    - New node always inserted at the beginning of the list

Building a Linked List

1. Creating the List in **Forward Direction**

Example: Input integers into a linked list where the input ends with -99

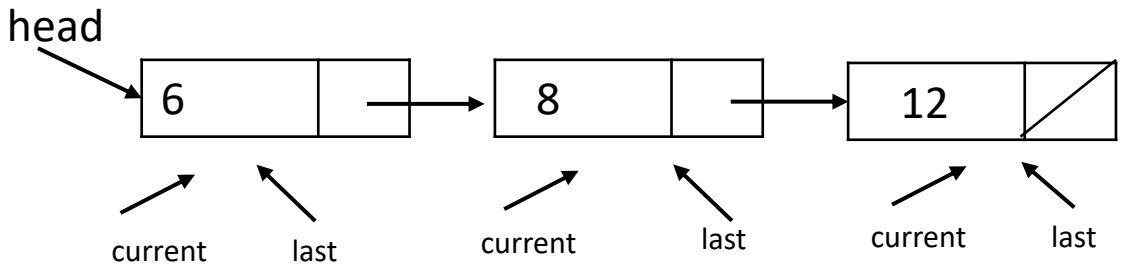
- 3 Pointers will be required: head, last and current.
- New node always inserted at end of the linked list

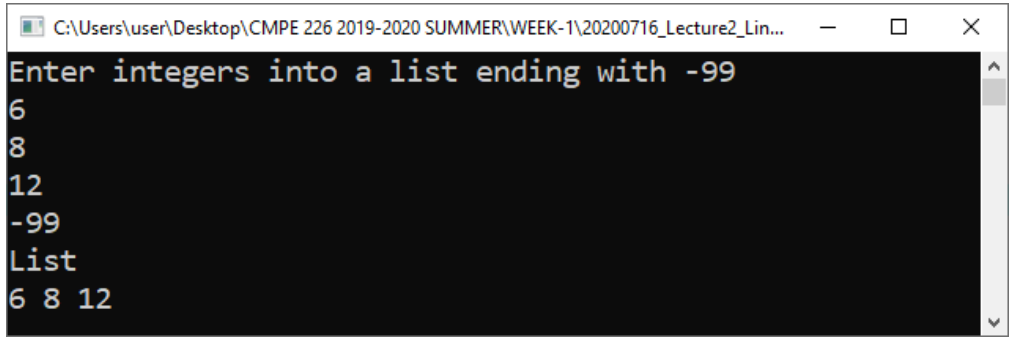
```
C:\Users\user\Desktop\CMPE 226 2019-2020 SUMMER\WEEK-1\20200716_Lecture2_Lin...
Enter integers into a list ending with -99
6
8
12
-99
List
6 8 12
```

```
struct Node{
    int info;
    Node *link;
};
Node* createForward(){

    Node *head, *last, *current;
    int  num;
    cout<<"Enter integers into a list ending with -99"<<endl;
    cin>>num;
    head=NULL;
    while(num!=-99){
        current = new Node;
        current->info=num;
        current->link=NULL;
        if(head==NULL){//empty list?
            head=current;
            last=current;
        }
        else{
            last->link=current;
            last=current;
        }
        cin>>num;

    }//end while
    return head;
}
```

Building a Linked List

1. Creating the List in **Forward Direction**






```
struct Node{
    int info;
    Node *link;
};
Node* createForward(){

    Node *head, *last, *current;
    int  num;
    cout<<"Enter integers into a list ending with -99"<<endl;
    cin>>num;
    head=NULL;
    while(num!=-99){
        current = new Node;
        current->info=num;
        current->link=NULL;
        if(head==NULL){//empty list?
            head=current;
            last=current;
        }
        else{
            last->link=current;
            last=current;
        }
        cin>>num;

    }//end while
    return head;
}
```

Building a Linked List: Creating the List in **Forward Direction**

```
#include<iostream>
using namespace std;
struct Node{
        int info;
        Node *link;
};
Node* createForward(){
        Node *head, *last, *current;
        int  num;
cout<<"Enter integers into a list ending with -99"<<endl;
        cin>>num;
        head=NULL;
        while(num!=-99){
                current = new Node;
                current->info=num;
                current->link=NULL;
If(head==NULL){//empty list?
                head=current;
                last=current;}
else{
                last->link=current;
                last=current;}
                cin>>num;
}//end while
return head;
}
int main(){
                Node *h,*t;
                h=createForward();
                t=h;
                cout<<"List"<<endl;
                while(t!=NULL){
                        cout<<t->info<<" ";
                        t=t->link;
                }
                return 0;
}
```

Building a Linked List

2. Creating the List in **Backward Direction**

Example: Input integers into a linked list where the input ends with -99

- Head and current pointers are used. (no need for last)
- New node always inserted at the beginning of the list

```
C:\Users\user\Desktop\CMPE 226 2019-2020 SUMMER\WEEK-1\20200716_Lecture2_Li...
Enter integers into a list ending with -99
2
3
15
28
-99
List:
28 15 3 2
```

```
struct Node{
    int info;
    Node *link;
};
Node* createBackwards(){

    Node *head, *current;
    int  num;
    cout<<"Enter integers into a list ending with -99"<<endl;
    cin>>num;
    head=NULL;
    while(num!=-99){
        current = new Node;
        current->info=num;
        current->link=head;
        head=current;
        cin>>num;
    }//end while
    return head;
}
```

Building a Linked List: Creating the List in **Backward Direction**

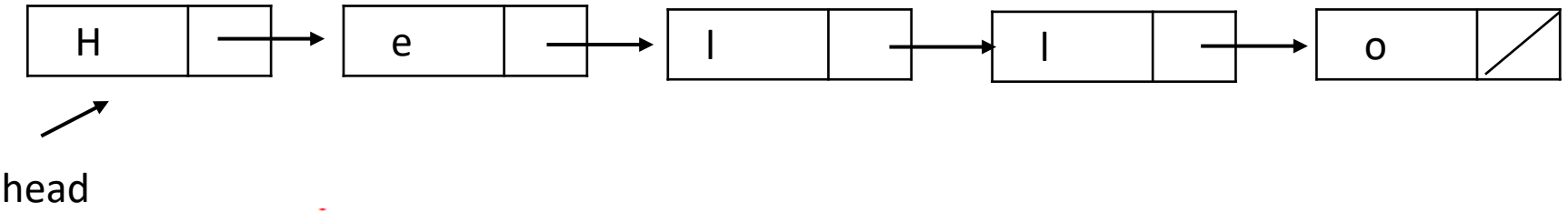
```
#include<iostream>
using namespace std;
struct Node{
        int info;
        Node *link;
};
```

```
Node* createBackwards(){
        Node *head, *current;
        int  num;
        cout<<"Enter integers into a list ending with -99"<<endl;
        cin>>num;
        head=NULL;
        while(num!=-99){
                current = new Node;
                current->info=num;
                current->link=head;
                head=current;
                cin>>num;
        }//end while
        return head;
}
```

```
int main()
        {
        Node *h,*t;
        h=createBackwards();
        t=h;
        cout<<"List:"<<endl;
        while(t!=NULL){
                cout<<t->info<<" ";
                t=t->link;
        }
        return 0;
}
```

Example: Search

Read a character, search it in a linked list. If found replace with ‘a’




```
//Search for the letter
cout<<endl<<"Enter a letter:";
cin>>c;
p=h;
while((p!=NULL) && (p->info!=c)){
    p=p->link;
}

if(p!=NULL)
{
    cout<<c<<" is found at node "<<endl;
    p->info='a';
}
else
    cout<<c<<" was not found."<<endl;
```

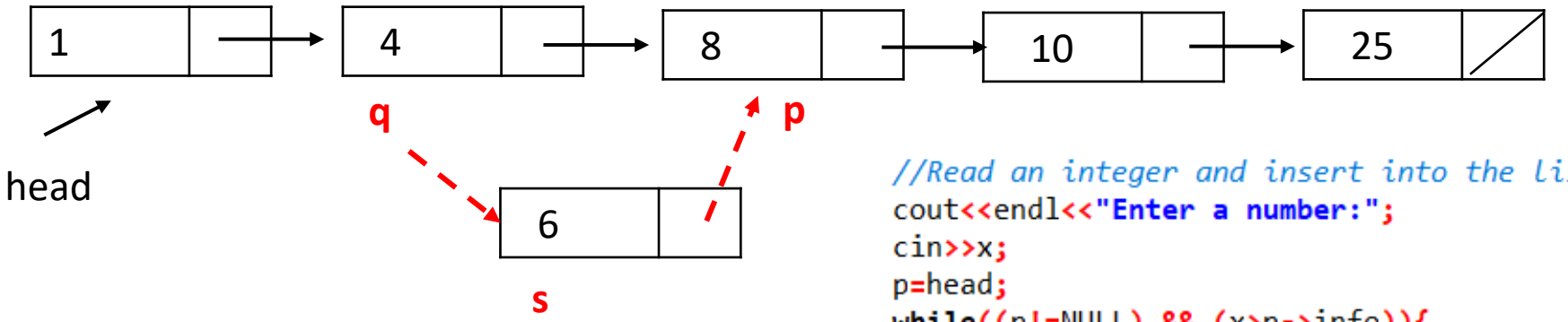
Example: Search a Character in a Linked List

```
#include<iostream>
using namespace std;
int main(){
Node *h,*t,*p;
char c;
//Create List
h=createForward();
t=h;
cout<<"List:"<<endl;
while(t!=NULL){
        cout<<t->info<<" ";
        t=t->link;
}
//Search for the letter
cout<<endl<<"Enter a letter:";
cin>>c;
p=h;
while((p!=NULL) && (p->info!=c)){
        p=p->link;
}
if(p!=NULL)
{
        cout<<c<<" is found at node "<<endl;
        p->info='a';
}
else
        cout<<c<<" was not found."<<endl;
//Print new list
p=h;
cout<<"New List:"<<endl;
while(p!=NULL){
cout<<p->info<<" ";
                p=p->link;
                }
return 0;
}
```

Example: Insert

Read an integer and insert it into its proper place in a linked list sorted in ascending order.

3 Pointers will be required: *s: to be inserted, *q: preceeding node, *p: following node




```
//Read an integer and insert into the list in ascending order
cout<<endl<<"Enter a number:";
cin>>x;
p=head;
while((p!=NULL) && (x>p->info)){
    q=p;
    p=p->link;
}
s=new Node;
s->info=x;
s->link=p;

if(p==head)
    head=s;
else
    q->link=s;
```

Example: Inserting a node into a linked list sorted in ascending order.

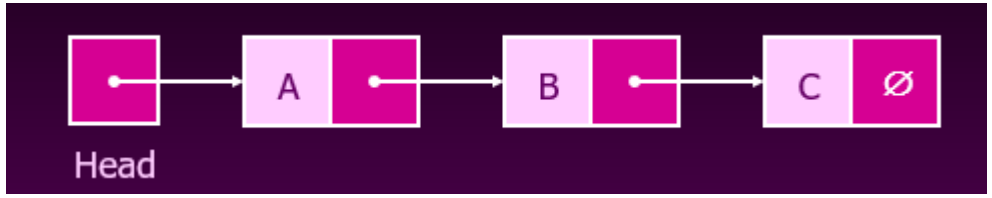
```
#include<iostream>
using namespace std;
int main(){
Node *head,*t,*p,*q,*s;
int x;
//Create List
h=createForward();
t=head;
cout<<"List:"<<endl;
while(t!=NULL){
        cout<<t->info<<" ";
        t=t->link;
}
```

```
//Read an integer and insert into the list in ascending order
cout<<endl<<"Enter a number:";
cin>>x;
p=head;
while((p!=NULL) && (x>p->info)){
                q=p;
                p=p->link;
                }
s=new Node;
s->info=x;
s->link=p;
if(p==head)
                head=s;
else
                q->link=s;
```

```
//Print new list
p=h;
cout<<"New List:"<<endl;
while(p!=NULL){
cout<<p->info<<" ";
                p=p->link;
                }
return 0;
}
```

Overview

- Linked List: A series of connected nodes
- Basic Operations:
  - Insert, Delete, Search, Print
- Variations of Linked Lists:
  - Singly Linked Lists
  - Circular Linked Lists
  - Doubly Linked Lists
- Linked Lists as an ADT



Discussion: Array versus Linked Lists

- 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.
  - **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.

Each data item in a linked list is created/deleted individually.

References

- CMPE226- Lecture Notes by Cigdem Turhan
- Data Structures Using C++, D.S. Malik, Thomson Course Technology, 2nd Edition.
- Lecture Slides by Huamin Qu, The Hong Kong University of Science and Technology (2005)