



# COMP 2012H Honors Object-Oriented Programming and Data Structures

## Topic 9: Linked List

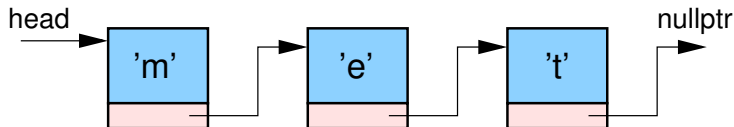
Dr. Desmond Tsoi

Department of Computer Science & Engineering  
The Hong Kong University of Science and Technology  
Hong Kong SAR, China



# What is a Linked List?

- A *list* is a linear sequence of objects.
- You may implement a list by an *array*. e.g. `int x[5];`
  - ▶ *Advantage*: array is an efficient data structure that works well with loops and recursion.
  - ▶ *Disadvantage*: size of the array is determined *in advance*.
- A *linked list* links objects together by pointers so that each object is pointing to the next object in the sequence (list).
  - ▶ *Advantage*: It is *dynamic*; it grows and shrinks to any size as you want at *runtime*.
  - ▶ *Disadvantage*:
    - ★ requires additional memory for the linking pointers
    - ★ takes more time to manipulate its items



# A Typical C++ Linked List Definition

- Each object in a linked list is usually called a “**node**”.
- The typical C++ definition for a **node** in a linked list is a **struct** (or

```
later class):  
    struct ll_node  
    {  
        <type> data;    // contains useful information  
        ll_node* next; // the link to the next node  
    };
```

- The **first** and the **last** node of a linked list always need special attention.
- For the **last node**, its **next** pointer is set to **nullptr** to tell that it is the end of the linked list.
- We need a pointer variable, usually called **head** to point to the **first node**.
- Once you get the **head** of the linked list, you get the **whole list**!

# Basic Operations of a Linked List

```
/* To create a node */
ll_node* p = new ll_node;

/* To access/modify the data in a node */
cout << p->data;
cout << (*p).data;

cin >> p->data;
p->next = nullptr;

/* To set up the head of a linked list */
ll_node* head = nullptr; // An empty linked list
head = p;                // head points to the node that p points to

/* To delete a node */
delete p;                 // Dangling pointer
p = nullptr;              // Reset the pointer for safety reason
```

## Example: Create the LL-String “met”

```
#include "ll_cnode.h"    /* File: ll_main.cpp */

int main()    // Create the LL-string "met"
{    // Create each of the 3 ll_cnodes
    ll_cnode* mp = new ll_cnode; mp->data = 'm';
    ll_cnode* ep = new ll_cnode; ep->data = 'e';
    ll_cnode* tp = new ll_cnode; tp->data = 't';

    // Hook them up in the required order to create the LL
    mp->next = ep;
    ep->next = tp;
    tp->next = nullptr;

    // Traverse the LL and print out the data sequentially
    for (ll_cnode* p = mp; p; p = p->next)
        cout << p->data;
    cout << endl;

    // Clean up
    delete mp; delete ep; delete tp; return 0;
}
```

# Common Operations on a Linked List

- Common operations:
  - ▶ **Create** a new linked list.
  - ▶ **Search** data in the list.
  - ▶ **Delete** a node in the list.
  - ▶ **Insert** a new node in the list.
- For all these operations, again special attention is usually needed when the operation involves the **first** or the **last** node.



## Example: LL-String — ll\_cnode.h

Let's use a linked list (instead of an array) of characters to represent a string.

---

```
#include <iostream> /* File: ll_cnode.h */
using namespace std;

struct ll_cnode
{
    char data;           // Contains useful information
    ll_cnode* next;      // The link to the next node
};

const char NULL_CHAR = '\0';
ll_cnode* ll_create(char);
ll_cnode* ll_create(const char []);
int ll_length(const ll_cnode*);
void ll_print(const ll_cnode*);
ll_cnode* ll_search(ll_cnode*, char c);
void ll_insert(ll_cnode*&, char, unsigned);
void ll_delete(ll_cnode*&, char);
void ll_delete_all(ll_cnode*&);
```

## Example: LL-String — ll\_create.cpp

```
#include "ll_cnode.h"    /* File: ll_create.cpp */
// Create a ll_cnode and initialize its data
ll_cnode* ll_create(char c)
{
    ll_cnode* p = new ll_cnode; p->data = c; p->next = nullptr; return p;
}

// Create a linked list of ll_cnodes with the contents of a char array
ll_cnode* ll_create(const char s[])
{
    if (s[0] == NULL_CHAR) // Empty linked list due to empty C string
        return nullptr;

    ll_cnode* head = ll_create(s[0]); // Special case with the head

    ll_cnode* p = head; // p is the working pointer
    for (int j = 1; s[j] != NULL_CHAR; ++j)
    {
        p->next = ll_create(s[j]); // Link current cnode to the new cnode
        p = p->next; // p now points to the new ll_cnode
    }

    return head; // The WHOLE linked list can be accessed from the head
}
```



## Example: LL-String — ll\_length.cpp, ll\_print.cpp

```
#include "ll_cnode.h" /* File: ll_length.cpp */

int ll_length(const ll_cnode* head)
{
    int length = 0;
    for (const ll_cnode* p = head; p != nullptr; p = p->next)
        ++length;
    return length;
}
```

```
#include "ll_cnode.h" /* File: ll_print.cpp */

void ll_print(const ll_cnode* head)
{
    for (const ll_cnode* p = head; p != nullptr; p = p->next)
        cout << p->data;
    cout << endl;
}
```

## Example: LL-String — ll\_search.cpp

```
#include "ll_cnode.h"    /* File: ll_search.cpp */

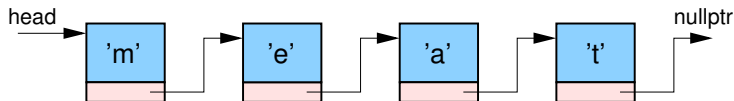
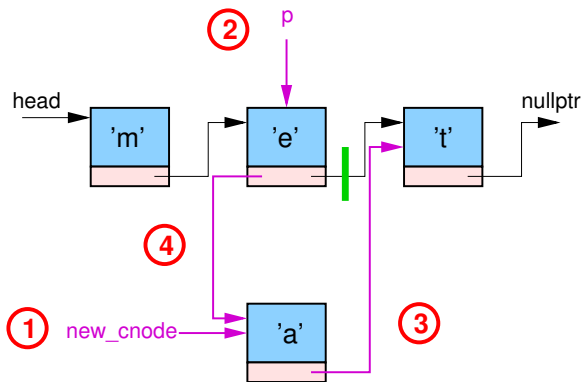
// The returned pointer may be used to change the content
// of the found ll_cnode. Therefore, the return type
// should not be const ll_cnode*.

ll_cnode* ll_search(ll_cnode* head, char c)
{
    for (ll_cnode* p = head; p != nullptr; p = p->next)
    {
        if (p->data == c)
            return p;
    }

    return nullptr;
}
```



## Example: LL-String — Insertion Algorithm



## Example: LL-String — ll\_insert.cpp I

```
#include "ll_cnode.h"    /* File: ll_insert.cpp */

// To insert character c to the linked list so that after insertion,
// c is the n-th character (counted from zero) in the list.
// If n > current length, append to the end of the list.

void ll_insert(ll_cnode*& head, char c, unsigned n)
{
    // STEP 1: Create the new ll_cnode
    ll_cnode* new_cnode = ll_create(c);

    // Special case: insert at the beginning
    if (n == 0 || head == nullptr)
    {
        new_cnode->next = head;
        head = new_cnode;
        return;
    }
}
```

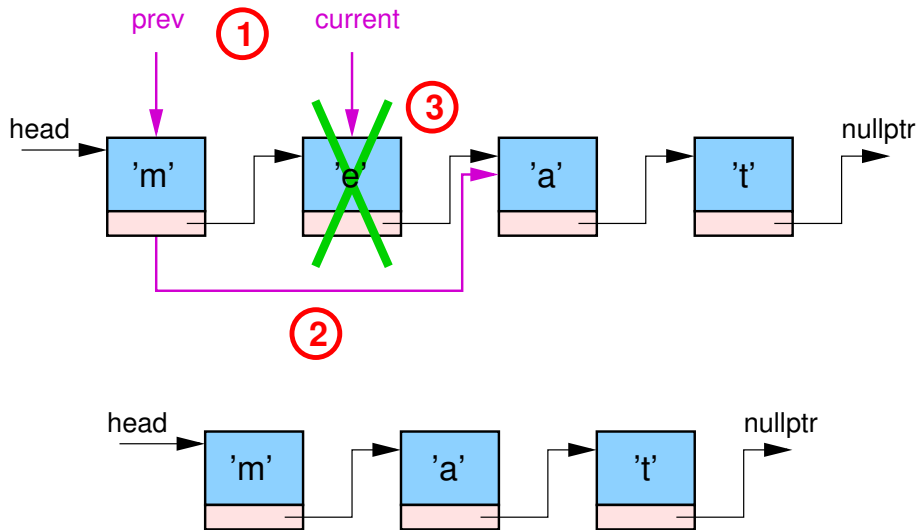
## Example: LL-String — ll\_insert.cpp II

```
// STEP 2: Find the node after which the new node is to be added
ll_cnode* p = head;
for (int position = 0;
    position < n-1 && p->next != nullptr;
    p = p->next, ++position)
    ;

// STEP 3,4: Insert the new node between
//           the found node and the next node
new_cnode->next = p->next; // STEP 3
p->next = new_cnode;      // STEP 4
}
```



## Example: LL-String — Deletion Algorithm



## Example: LL-String — ll\_delete.cpp

```
#include "ll_cnode.h"    /* File: ll_delete.cpp */
// To delete the character c from the linked list.
// Do nothing if the character cannot be found.
void ll_delete(ll_cnode*& head, char c)
{
    ll_cnode* prev = nullptr; // Point to previous ll_cnode
    ll_cnode* current = head; // Point to current ll_cnode

    // STEP 1: Find the item to be deleted
    while (current != nullptr && current->data != c)
    {
        prev = current;          // Advance both pointers
        current = current->next;
    }

    if (current != nullptr) // Data is found
    {
        // STEP 2: Bypass the found item
        if (current == head) // Special case: delete the first item
            head = head->next;
        else
            prev->next = current->next;

        delete current;          // STEP 3: Free up the memory of the deleted item
    }
}
```

## Example: LL-String — ll\_delete\_all.cpp

```
#include "ll_cnode.h"    /* File: ll_delete_all.cpp */

// To delete the WHOLE linked list, given its head by recursion.
void ll_delete_all(ll_cnode*& head)
{
    if (head == nullptr) // An empty list; nothing to delete
        return;

    // STEP 1: First delete the remaining nodes
    ll_delete_all(head->next);

    // For debugging: this shows you what are deleting
    cout << "deleting " << head->data << endl;

    delete head;          // STEP 2: Then delete the current nodes
    head = nullptr;       // STEP 3: To play safe, reset head to nullptr
}
```



## Example: LL-String — ll\_test.cpp I

```
#include "ll_cnode.h"    /* File: ll_test.cpp */
int main()
{
    ll_cnode* ll_string = ll_create("met");
    cout << "length of ll_string = " << ll_length(ll_string) << endl;
    ll_print(ll_string);
    ll_print(ll_search(ll_string, 'e'));

    cout << endl << "After inserting 'a'" << endl;
    ll_insert(ll_string, 'a', 2); ll_print(ll_string);
    cout << endl << "After deleting 'e'" << endl;
    ll_delete(ll_string, 'e'); ll_print(ll_string);
    cout << endl << "After deleting 'm'" << endl;
    ll_delete(ll_string, 'm'); ll_print(ll_string);

    cout << endl << "After inserting 'e'" << endl;
    ll_insert(ll_string, 'e', 9); ll_print(ll_string);
    cout << endl << "After deleting 't'" << endl;
    ll_delete(ll_string, 't'); ll_print(ll_string);
    cout << endl << "After deleting 'e'" << endl;
```

## Example: LL-String — ll\_test.cpp II

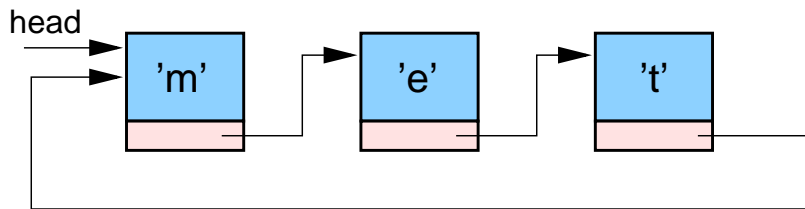
```
ll_delete(ll_string, 'e'); ll_print(ll_string);
cout << endl << "After deleting 'a'" << endl;
ll_delete(ll_string, 'a'); ll_print(ll_string);

cout << endl << "After deleting 'z'" << endl;
ll_delete(ll_string, 'z'); ll_print(ll_string);
cout << endl << "After inserting 'h'" << endl;
ll_insert(ll_string, 'h', 9); ll_print(ll_string);
cout << endl << "After inserting 'o'" << endl;
ll_insert(ll_string, 'o', 0); ll_print(ll_string);

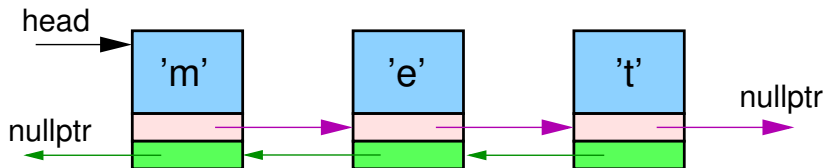
ll_delete_all(ll_string);
return 0;
}
```

# Other Common Variants of Linked List

- Circular Linked List



- Doubly Linked List



That's all!

Any questions?

