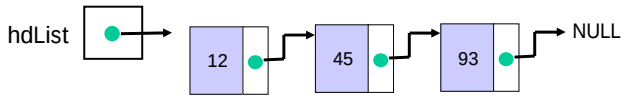
Ordered Lists

Lecture 6

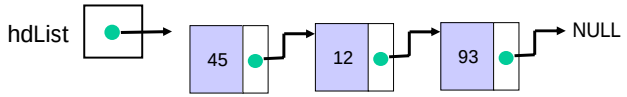
- This lecture presents ***ordered lists***. An ordered list is one which is maintained in some predefined order, such as alphabetical or numerical order.
- We'll study a linked list implementation.
- We need routines that can,
  - insert a new element, maintaining the order, and
  - delete an element from the list.
- We'll also consider ***lookup***, a routine to extra information from the list.

Ordered Lists Defined

- A list is numerically ordered if, for every item X in the list, every item after X in the list is greater than X or equal to X.
- So the following list is ordered (in ascending order).

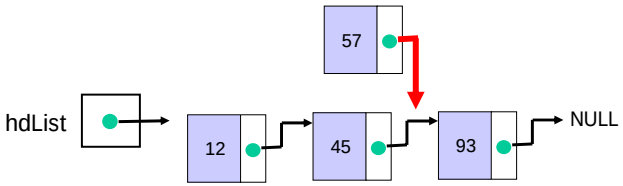


- But the following list is not.



Insertion into an Ordered List I

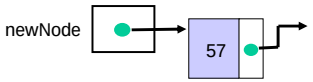
- Suppose we want to insert **57** into the list, maintaining its ascending order.
- We search along the list until we find the first item which is greater than **57**.
- This is the 3rd entry on the list, i.e. the entry for **93**.



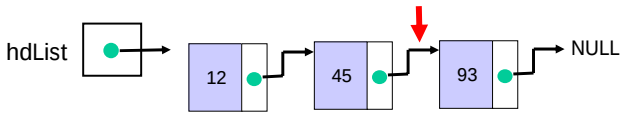
- We want to insert **57** **before** this item.

Insertion into an Ordered List II

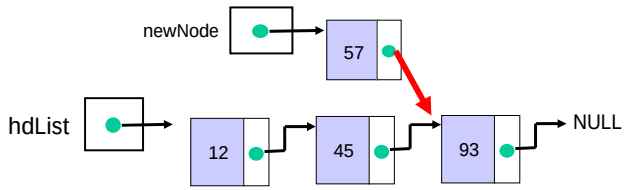
- Step 1: we create a new node for **57**



- Step 2: Identify where to insert (before **93**)

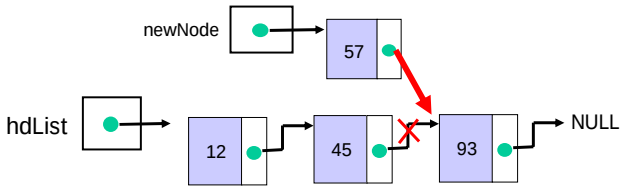


- Step 3: Make **57** point to **93**

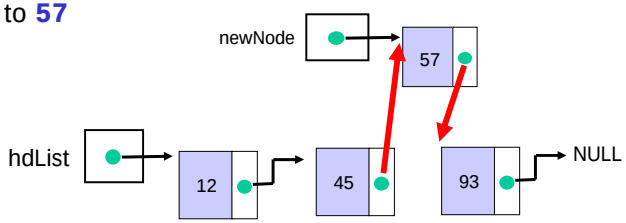


Insertion into an Ordered List II

- Step 4: Break the link between **45** and **93**

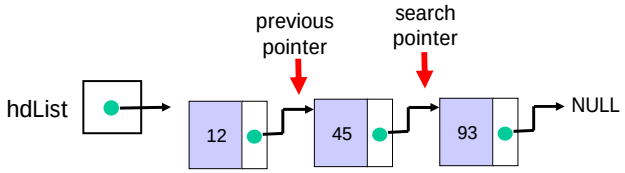


- Step 5: Re-link it to **57**



Insertion into an Ordered List III

- This looks straightforward. But to write a function that does this needs a little more thought.
- For a start, once we've found the first item greater than 57 (i.e. 93), we've already moved past 45, whose entry we want to modify.
- So we maintain two pointers in the search. The *search pointer* itself, plus a *previous pointer*, which is one item behind.



Insertion into an Ordered List IV

- Another problem is that we have to treat it as a special case if we need to insert the new item as the *first* element in the list.
- This is the only case in which the pointer to the whole list is modified.
- The old list will be passed to the insertion routine as a call-by-reference parameter. This special case is the only time this parameter is actually modified by the procedure.

Ordered Lists as an ADT

- We'll use the same type declaration as before.
- And we'll write two access routines `insertItem()` and `deleteItem()`.

```
typedef int Item;

class Node {
    public:
      Item data;
      Node* next;
};

typedef Node* NodePtr;

NodePtr hdList = NULL;

void insertItem (Item    data,
                 NodePtr &hdList){
    bool found = FALSE;
    NodePtr searchPtr, lastPtr, newPtr;
```

A new getItem() function

- In the previous lecture, the integers in the list was generated randomly. This is no help to us to in building an ordered list because we have no control over what numbers are generated.
- We will use a new `getItem()` function which read numbers from the keyboard. `insertItem()` and `deleteItem()`.

```
Item getItem () {
    int number;
    char str[80];
    cout << "Enter an integer: ";
    cin >> str;
    number = atoi(str);
    return Item(number);
}
```


Insertion

We first create the new node.

Must handle empty list as a special case.

Insert at the head of the list is also a special case.

```
void insertItem (Item    data,
                 NodePtr &hdList){
    bool found = FALSE;
    NodePtr searchPtr, lastPtr, newPtr;

    // create node
    newPtr = new Node;
    newPtr->data = data;
    newPtr->next = NULL;

    // empty list is special case
    if (hdList==NULL) {
        hdList = newPtr;
        return;
    }
    // insert at head is speical too
    else if (hdList->data >= data) {
        newPtr->next = hdList;
        hdList = newPtr;
    }
```


Insertion - normal case

Initialize things

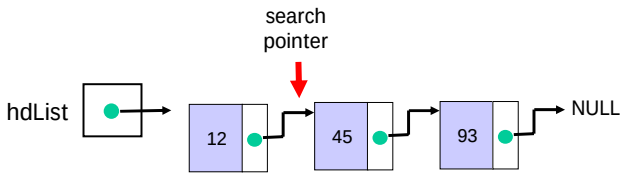
Search to find where to insert item

Re-link item into the correct place

```
    // normal insertion
    else {
        found = FALSE;
        searchPtr = hdList;
        lastPtr = hdList;
        while ((searchPtr != NULL) && (! found)) {
            if (searchPtr->data >= data)
                found = TRUE;
            else {
                lastPtr = searchPtr;
                searchPtr = searchPtr->next;
            }
        }
        newPtr->next = searchPtr;
        lastPtr->next = newPtr;
    }
```

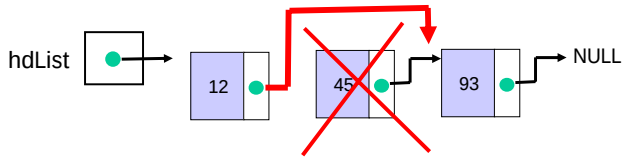

Deletion from an Ordered List I

- Deletion is similar to insertion.
- Suppose we want to delete **45** from the original list
- First search along the list until we find **45**. (If **45** isn't there, we've finished.)



Deletion from an Ordered List II

- Then **45** is detached from the list by linking **12** to **93** directly.
- We mustn't forget to return the space used up by **45** to the free area of the heap.



- As with insertion, we need to maintain both a search pointer and a previous pointer.
- As with insertion, we need to treat deletion of the first element of the list as a special case.

Deletion from an Ordered List III

- As with insertion, we need to maintain both a search pointer and a follow pointer.
- As with insertion, we need to treat deletion of the first element of the list as a special case.

Deletion

Empty list is special: do nothing!

Delete from head of list is also special.

Initialize things.

Search for item.

Found here, delete!

```
void deleteItem (Item data, NodePtr &hdList){
    bool found = FALSE;
    NodePtr searchPtr, lastPtr, oldPtr;
    // empty list is special case
    if (hdList==NULL) {
        return;
    }
    else if (hdList->data == data) {
        oldPtr = hdList;
        hdList = hdList->next;
        delete oldPtr;
    }
    else {
        found = FALSE;
        searchPtr = hdList;
        lastPtr = hdList;
        while ((searchPtr != NULL) && (! found)) {
            if (searchPtr->data == data) {
                found = TRUE;
                lastPtr->next = searchPtr->next;
                delete searchPtr;
            }
            else {
                lastPtr = searchPtr;
                searchPtr = searchPtr->next;
            }
        }
    }
}
```


Lookup and its Uses

- Lists are useful for *looking things up*. Ordered lists are a little more efficient for this than unordered ones. (But not much.)
- Obvious examples of programs that need to look things up include:
  - the address book in your personal organiser,
  - the search facility on a CD ROM,
- But there are lots of less obvious uses too. A compiler has to maintain a *look-up table* for every variable name, every procedure name, etc, that you introduce in a program.
- If you understand insertion and deletion, lookup is just really a search in the list and return the required information.

What to do before the next lecture?

- Complete Exercise D (non-assessed) – See next slide
- Read all of Chapter 15 of Savitch

Ordered Linked list

Exercise D

- You should try the ordered list program given to you in the lecture and make sure that you understand how it works.
- Write the function `lookup()`, which has as input the number to search in an ordered list. It should then return the pointer to that particular item.