DVA104

Data Structures, Algorithms, and Program Development

Abu Naser Masud

masud.abunaser@mdh.se

Contribution: Gabriele Capannini
Caroline Uppsäll

Mälardalen University

September 17, 2018

Reading Lists

- Pointers and Memory (click to go to the page)
- Linked Lists: Introduction to Algorithms (Cormen) (Pages: 236-245)
- Data structure using C, Reema Thareja, PDF is here (unsafe link) Pointers: PP 34-38, Linked Lists: PP 162-218

Recap: Pointers and Memory

- A pointer stores the memory address of another memory location.
- Obtaining the value stored in a pointer reference is called *dereferencing* the pointers.
- NULL pointer points to "nothing". Dereferencing a NULL pointer may crash your program!!!

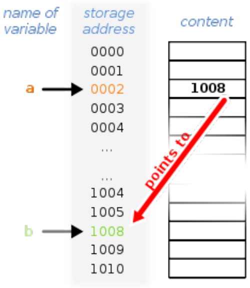


Figure: *

$$int * a = \&b;$$

(Fig: wikipedia)

Recap: Pointers and Memory

What happens when we have assignments between pointers?
For example:

```
int *a,*c,b=10;
a=&b;
c=a;
```

Variable	Address	Contents
a	0002	1008
b	1008	10
c	1016	?

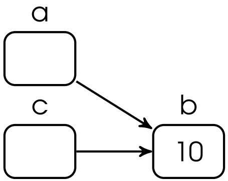
Recap: Pointers and Memory

What happens when we have assignments between pointers? For example:

```
int *a,*c,b=10;
a=&b;
c=a;
```

Variable	Address	Contents
a	0002	1008
b	1008	10
c	1016	1008

Graphically,



Recap: Pointers and Memory

Variable	Address	Contents
a	0002	1008
b	1008	10
c	1016	1008
p	1032	0002

What does p points to?

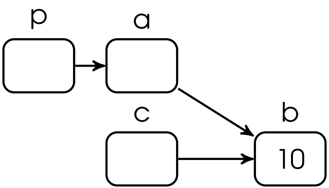
Recap: Pointers and Memory

Variable	Address	Contents
a	0002	1008
b	1008	10
c	1016	1008
p	1032	0002

```c
int **p,*a,*c,b=10;
a=&b;
c=a;
p=&a;
```

What does p points to?

p is a DOUBLE POINTER which points to another pointer.
Graphically,



Recap: Pointers and Memory

Application of double pointer (Example from stack exchange)

```
void alloc2(int** p) {
   *p = (int*)malloc(sizeof(int));
   **p = 10;
}

void alloc1(int* p) {
   p = (int*)malloc(sizeof(int));
   *p = 10;
}

int main(){
   int *p;
   alloc1(p);
   //printf("%d ",*p);//value is undefined
   alloc2(&p);
   printf("%d ",*p);//will print 10
   free(p);
   return 0;
}
```

Linked Lists

- We will go through a new data type: **linked list**.
- A linked list is an ADT (implementation of a list) and, at the same time, a specific data type used for the implementation of other ADTs
- A list is an alternative to a dynamic array, with its own pros and cons.

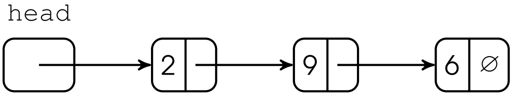
Arrays

- Advantages:
  - Direct access to the elements (position can be computed quickly).
  - Quick and easy to allocate and free.
  - Easy to use.
- Disadvantages:
  - Fixed size: arrays can be extended by allocating a new array and copying the original one.
  - Adding an element between two existing ones (by preserving the order) leads to expensive copies.

Singly Linked Lists

Linked Lists (LL) work like a chain: each ring is connected to the neighbor-rings. In the **Singly Linked List** case:

- Each element has only one link to the next element.
- The list is referred by means of the 1st ring (so-called head).



In C:

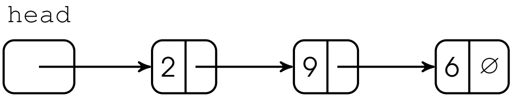
```
typedef int datatype;
struct node {
  datatype data;
  struct node* next;
};
typedef struct node Node;
struct node* head;
```

Linked Lists

- Advantages:
  - Elements can be inserted anywhere: front, back, and middle one.
  - LL can grow and shrink freely, no copies required.
  - LL use the exact amount of memory needed (even if each element requires to store a pointer, at least).
- Disadvantages:
  - Slower access to items in the middle of the list (need to scan)
  - LL are more difficult to implement.
  - LL may be slower in to add many elements to.

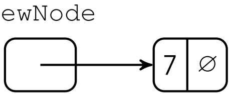
Read carefully `https://en.wikipedia.org/wiki/Linked_list`

Singly LL - add an item in front



We need a function to `CreateNewNode` to allocate the memory and initialize the new node, `newNode`:

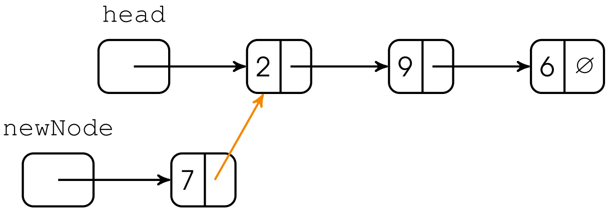
```
Node * newNode = createNewNode(7);
```



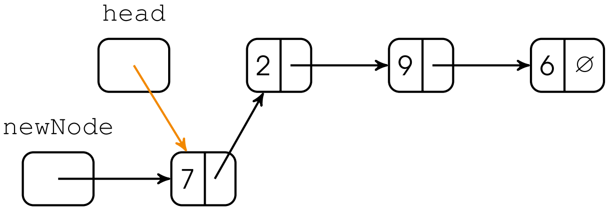
Singly LL - add an item in front (continued)

Now, we need to connect `newNode` to the existing list. Since we are adding `newNode` to the front of the list, the field `next` of `newNode` must point to the first item in the list as `head` does.

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



Singly LL - add an item in front (continued)



To keep consistent the list, even `head` must be updated:

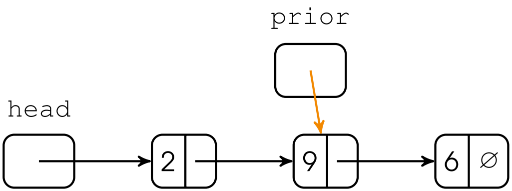
```
head = newNode;
```

Singly LL - add an item not in front

First, we need a pointer (`prior`) to the node after which we want to insert the new node. How we find it depends on what we have for criteria:

We can point to the correct node, for example, by scanning the list until a given condition is no more true. Example:

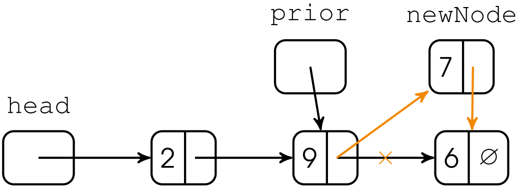
```
struct node* prior = head;
while (prior!=NULL && prior->data!=9)
  prior = prior-> next;
```



Singly LL - add an item not in front (continued)

When `prior` has been found we can link it to the list by updating the pointers:

```
Node * newNode = createNewNode(7);
newNode->next = prior->next;
prior->next = newNode;
```



Singly LL - add an item not in front (test)

Try to figure out what happen if we invert the order in updating the pointers; that is. . . If we do this:

```
Node * newNode = createNewNode(7);
prior->next = newNode;
newNode->next = prior->next;
```

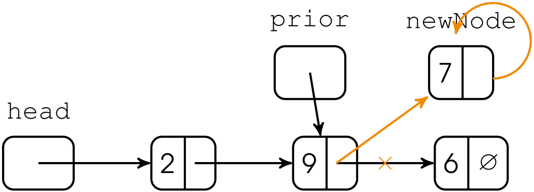
instead of this:

```
Node * newNode = createNewNode(7);
newNode->next = prior->next;
prior->next = newNode;
```

Singly LL - add an item not in front (test)

```
Node * newNode = createNewNode(7);
prior->next = newNode;
newNode->next = prior->next;
```

The code above generates this situation:



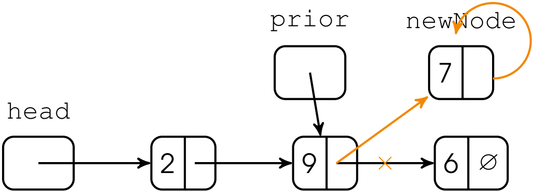
As a consequnce:

- the tail of the list (i.e., all nodes after '9' in the original list) is lost,
- a self-pointing node leads to a neverending execution in a loop scanning the list.

Singly LL - add an item not in front (test)

```
Node * newNode = createNewNode(7);
prior->next = newNode;
newNode->next = prior->next;
```

The code above generates this situation:



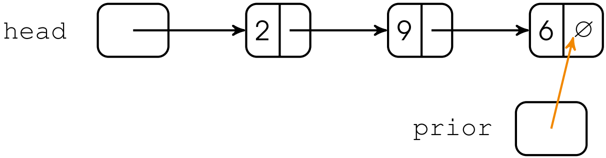
As a consequnce:

- the tail of the list (i.e., all nodes after '9' in the original list) is lost,
- a self-pointing node leads to a neverending execution in a loop scanning the list.

Singly LL - add an item <u>not</u> in front (test)

```
struct node* prior = head;
while (prior!=NULL && prior->data!=X)
                prior = prior-> next;
Node * newNode = createNewNode(7);
prior->next = newNode;
newNode->next = prior->next;
```

What if *X* is not within the list? The above code has NULL pointer dereference.



Singly LL - add an item not in front (test)

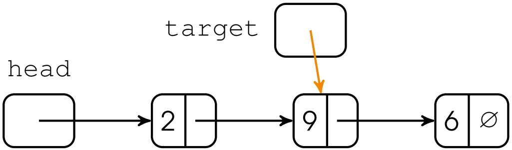
The correct code will be

```
struct node* prior = head;
while (prior!=NULL && prior->data!=X)
            prior = prior-> next;
Node * newNode = createNewNode(7);
if(prior!= NULL){
        prior->next = newNode;
        newNode->next = prior->next;
 }
```

Singly LL - remove an item not in front

First we find to the target node to delete (e.g., '9'):

```
struct node* target = head;
while (target!=NULL && target->data!=9)
  target = target-> next;
```



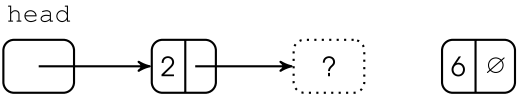
Can we remove the node as it follows?

```
free(target);
target = NULL;
```

Singly LL - remove an item not in front (continued)

```
free(target);
target = NULL;
```

The code above generates the following situation:

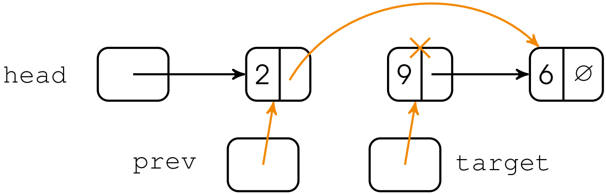


To remove the node '9' we need to know the item preceding the node '9'!

Singly LL - remove an item not in front (continued)

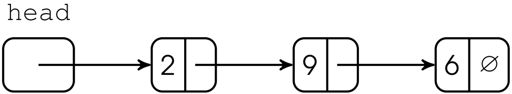
```
  struct node* target = head;
  struct node* prev = target; //extra pointer
  while(target!=NULL && target->data!=9) {
    prev = target;
    target = target->next;
  }
 if(target!=NULL){
  prev->next = target->next;
  free(target);
 }
```

The code above generates the following (correct) list:



Singly LL - print a list (test)

Given the following list:

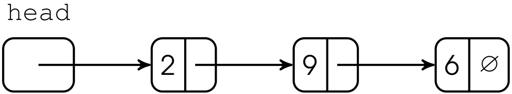


What do we get as result from the following code?

```
Node *current = head;
while(current->next!=NULL) {
  printf(" %d", current->data);
  current = current->next;
}
```

Singly LL - print a list (test)

Given the following list:



Since we test `current->next`, the last node '6' is not printed!

```
Node *current = head;
while(current->next!=NULL) {
  printf(" %d", current->data);
  current = current->next;
}
```

Singly LL - print a list (test)

Instead of this:

```
Node *current = head;
while(current->next!=NULL) {
  printf(" %d", current->data);
  current = current->next;
}
```

. . . We should use this:

```
Node *current = head;
while(current!=NULL) {
  printf(" %d", current->data);
  current = current->next;
}
```

Now the entire list will be shown.

Some examples of how to represent a list (1/3)

```
typedef Node* List;
List head;
```

- 👍 Simple implementation.
- 👍 Recursive properties: `head->next` is also a list.
- 👎 Requires the use of double pointer.
- 👎 Concept of node is ‘exposed’.

Some examples of how to represent a list (2/3)

```
struct list {
  struct Node* head;
};
typedef struct list List;
List myList;
```

- Obtains two different data types for node and list, allowing to include additional information (e.g., the length of the list).
- Saves double pointer in list functions.
- Two data types to keep track of.
- Losses recursive properties: `myList.head->next` is not a list.

Some examples of how to represent a list (3/3)

```
struct list {
  struct Node* head;
  struct Node* tail; //point to the last element
};
typedef struct list List;
List myList;
```

- 👍 Faster access to the last item. Try to design a FIFO (First-In, First-Out) LL: in this way you do not need to scan the list.
- 👎 Must update two pointers in add/remove operations.

LL as recursive data structure

By keeping the first type of representation:

```
struct node {
  int data;
  struct node* next;
};
typedef struct node Node;
typedef Node* List;
List list;
```

it's easy to see that `List` represents at the same time:

- The base case (`list`).
- The recursive case, since each `next` field is the same type of `list`.

LL as recursive data structure (test)

What is the following function for?

```
typedef Node* List;

int foo(List list){
  if(list!=NULL)
    return foo(list->next)+1;
  else
    return 0;
}
```

LL as recursive data structure

Other operations that work well to implement with recursion:

- Add/remove an item to the list (at a specific point or at the end).
- Print the list.
- Free memory for the entire list.
- Search for a data in the list.

In other words, all the functions requiring to iterate the list. For each of them, we have two cases:

- The list is empty (`NULL`).
- The list is a node with `data` and a list (i.e., the field `next`).

LL as recursive data structure - example

Write a recursive function `pushBack` that inserts an element to the end of the list.

Let `pushFront` be a function to add an item in front to a list:

```
void pushFront(List* plist, int data) {
  Node * newNode = createNewNode(data);
  newNode->next = *plist;
  *plist = newNode;
}
```

The target function is:

```
void pushBack(List* plist, int data) {
  if(*plist==NULL)
    /* base case: list-end reached */
    pushFront(plist, data);
  else
    /* recursive case: move ahead */
    pushBack(&(*plist)->next, data);
}
```

LL as recursive data structure - example (continued)

Did you notice the double pointer?
`List` is defined as a pointer to `Node` which is a struct:

```
typedef Node* List;
```

Hence, `List* plist` is double pointer since it points to `List`:

```
void pushBack(List* plist, int data) ...
```

Now, pay attention to the precedence of C operators. Let us analyze the line of code related to the recursive case:

```
pushBack(&(*plist)->next, data);
```

`&    ( *    plist    )->next`

`plist` is a double pointer

`*plist` points to the current list

`(*plist)->next` points to the list starting from next element

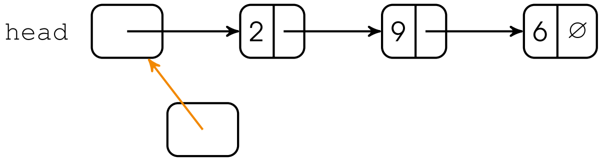
with `&`, we generate a pointer to the list pointed by `next`

LL as recursive data structure - example (continued)

Assume that we have a list like the one below, referred by `head` of which type is `List` (i.e., a pointer to `Node`). Then, in the `main` function we call:

```
pushBack(&head, 3);
```

In the function `pushBack`, `plist` equals `&head` and points to `head`:



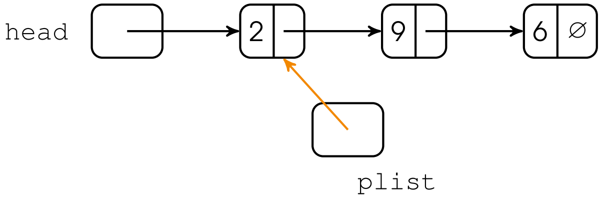
`plist` of which type is `List*` (double pointer)

LL as recursive data structure - example (continued)

Within the function `pushBack`, after the first recursive call:

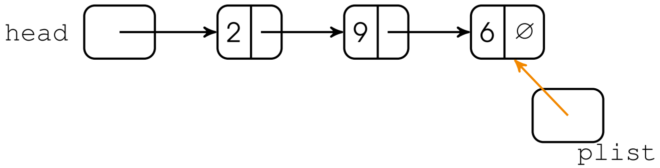
```
pushBack(&(*plist)->next, data);
```

`plist` points to the field `next` of the first element in the list (i.e., '2'). This is a property of recursion, since `head` and `next` are of the same type, i.e., list pointers:

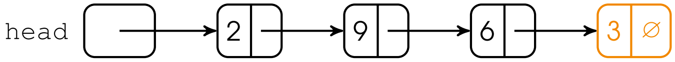


LL as recursive data structure - example (continued)

The recursion in the function `pushBack` continues until we reach the last node and `*plist==NULL`



Now the function `pushFront` is called and the new element is added at the end of the list.

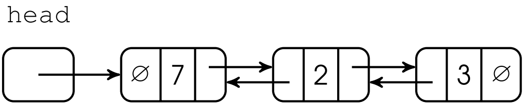


Double LL

```
struct node {
  int data;
  struct node* prev;
  struct node* next;
};
typedef struct node Node;
```

Example:

```
typedef Node* List;
List head;
```



Double LL

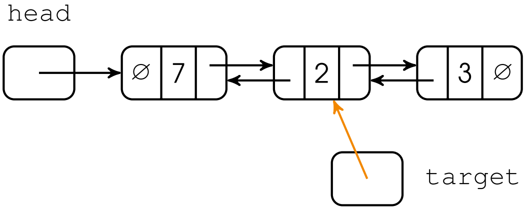
A **Double-linked List** has the following pros and cons when compared to a Singly LL:

- We can scan the elements both forward and backward.
- Some operations (like removing nodes) are easier to implement.

- More links to keep track of implementation.
- Require more memory (one more pointer for every node).

Double LL - test

Let's see how to remove an item in the middle of a Double LL:

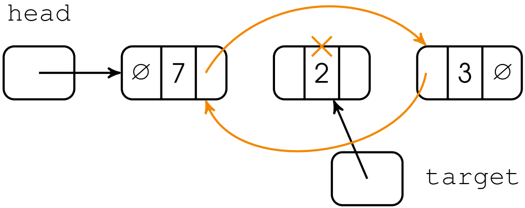



```
target->prev->next = target->next;
target->next->prev = target->prev;
free(target);
```

Does it work?

Double LL - test (continued)

Yes! It works:



There is no needs for extra pointers since all the information about the memory position of neighbors are stored in the `target` node (not only the position of the `next` node).