Advanced Uses of Pointers (3)
Linked Lists

Program Design (II)

2022 Spring

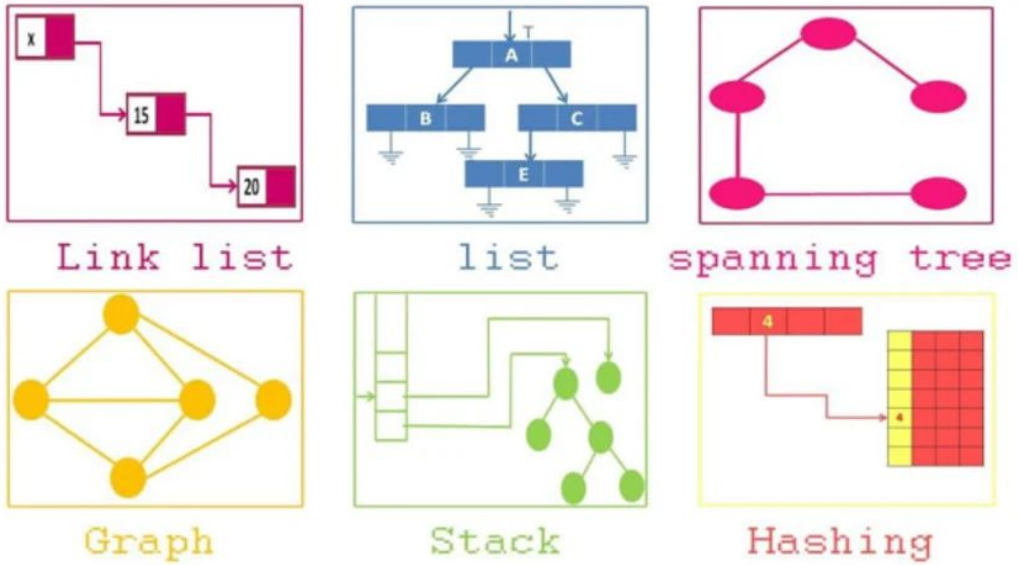
Fu-Yin Cherng
Dept. CSIE, National Chung Cheng University

Outline

- Introduce Linked Lists
- Common Operations of Linked Lists

Dynamic Storage Allocation

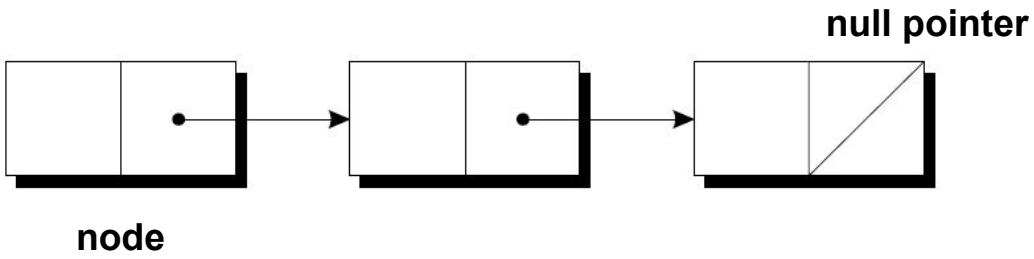
- Dynamic storage allocation is especially useful for building lists, trees, graphs, and other linked data structures.



ref. DEV Community

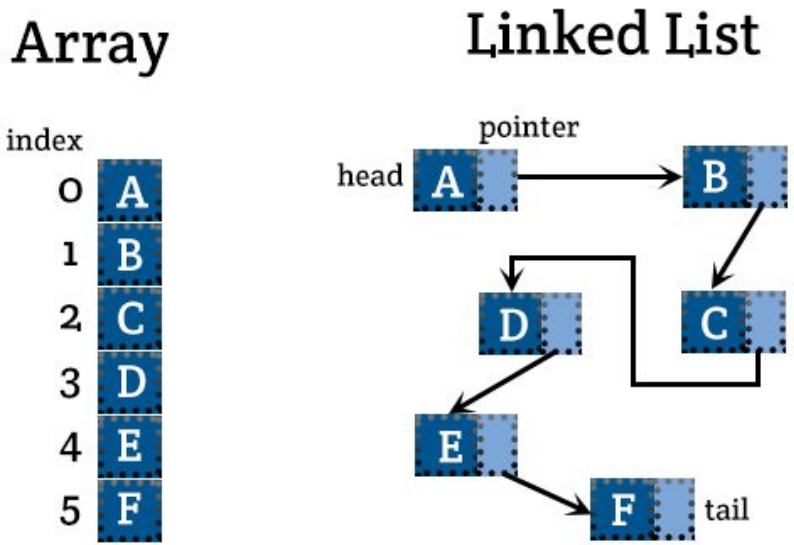
Linked Lists

- Linked lists is one of the most **commonly** used data structures.
- A ***linked list*** consists of a chain of structures (called ***nodes***), with each node containing a pointer to the next node in the chain
- The last node in the list contains a null pointer (diagonal line).



Linked Lists

- A linked list is more **flexible** than an array!
- we can easily **insert** and **delete** nodes in a linked list, allowing the list to grow and shrink as needed.
- Any element of an array can be accessed in the **same amount** of time.
- Accessing a node in a linked list is **fast** if the node is close to the beginning of the list, **slow** if it's near the end.



ref. Open4Tech

Linked Lists

- How to set up a linked list in C
  - Declaring a Node Type
  - Creating a Node
- Common operations:
  - inserting a node at the beginning of a (linked) list
  - searching for a node
  - deleting a node

Declaring a Node Type

- To set up a linked list, we'll need a structure that represents a single node.
- A node structure will contain data plus a pointer to the next node in the list
- For simplicity, we have a basic node contain an integer and a pointer to net node

```
struct node {
      int value;           /* data stored in the node  */
      struct node *next;   /* pointer to the next node */
};
```

Declaring a Node Type

- Notice that the `next` member has type `struct node *`, which means that it can stor a pointer to a `node` structure
- `node` must be a tag, not a `typedef` name, or there would be no way to declare the type of `next`.

```
struct node {
      int value;           /* data stored in the node  */
      struct node *next;   /* pointer to the next node */
};
```

Declaring a Node Type

- Next, we'll need a variable that always points to the **first node** in the list
- Setting `first` to `NULL` indicates that the list is **initially empty**.

```
struct node {
      int value;           /* data stored in the node  */
      struct node *next;   /* pointer to the next node */
};
…
struct node *first = NULL;
```

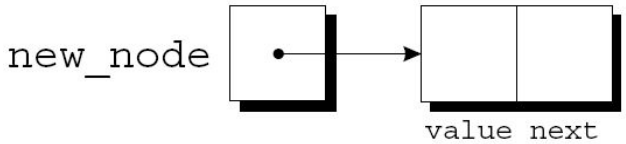
Creating a Node

- As we construct a linked list, we'll create nodes one by one, adding each to the list.
- Steps involved in creating a node:
  1. **Allocate memory for the node.**
  2. **Store data in the node.**
  3. Insert the node into the list.
- We'll concentrate on the first two steps for now.

Creating a Node

- When we create a node, we'll need a variable that can point to the node temporarily
- Use `malloc` to **allocate** memory for the new node, saving the return value in `new_node`
- `new_node` now **points** to a block of memory just large enough to hold a `node` structure

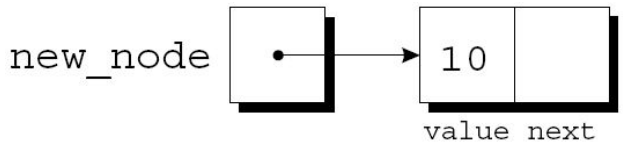
```
struct node *new_node;
new_node = malloc(sizeof(struct node));
```



Creating a Node

- Next, we'll **store data** in the `value` member of the new node
- We can **access** the member `value` via `new_node`
  - strucuter pointer (review)

```
struct node *new_node;
new_node = malloc(sizeof(struct node));
new_node->value = 10;
// or (*new_node).value = 10;
```

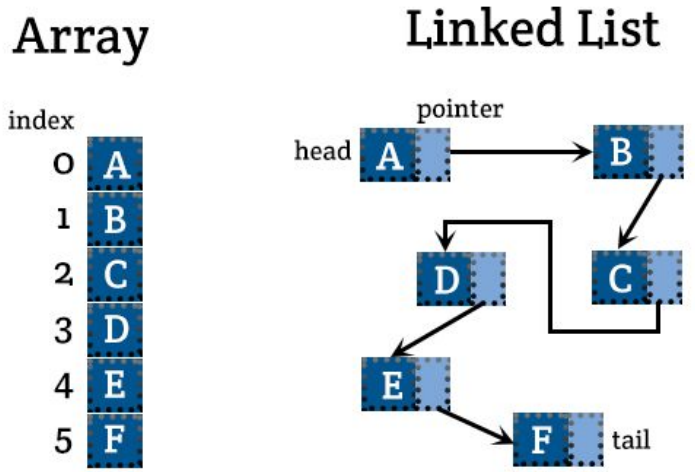


Inserting a Node at the Beginning of a Linked List

- Steps involved in creating a node:
  1. Allocate memory for the node.
  2. Store data in the node.
  3. **Insert the node into the list.**
- Common Operations
  1. **inserting a node at the beginning of a (linked) list**
  2. searching for a node
  3. deleting a node

Inserting a Node at the Beginning of a Linked List

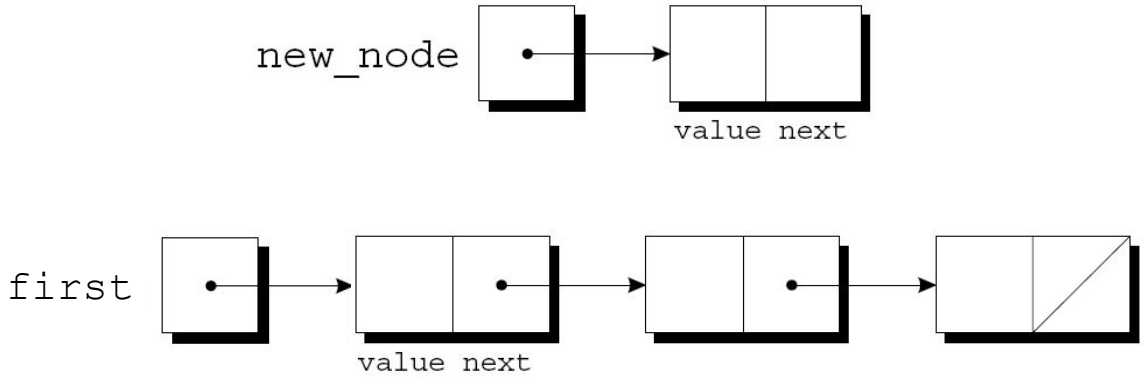
- One of the advantages of a linked list is that nodes can be added at any point in the list.
- However, the **beginning** of a list is the easiest place to insert a node.
- So, let's focus on this case first



ref. Open4Tech

Inserting a Node at the Beginning of a Linked List

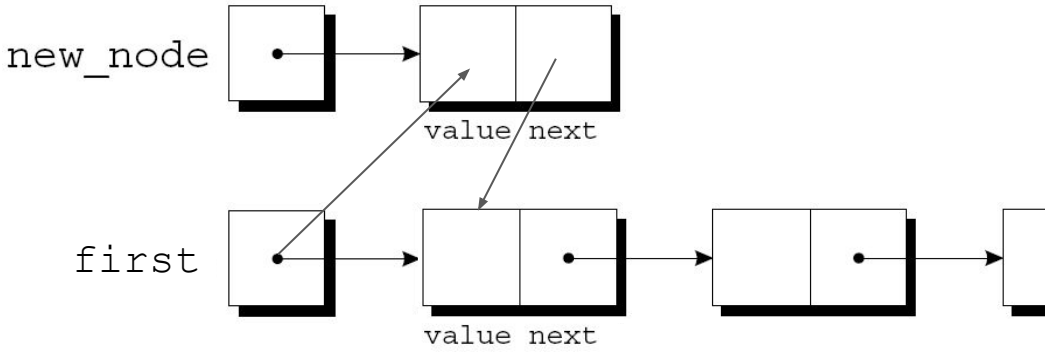
- Suppose that `new_node` is pointing to the node to be inserted, and `first` is pointing to the first node in the linked list.
- It takes **two statements** to insert the node into the list.



Inserting a Node at the Beginning of a Linked List

- The **first** step is to **modify** the **new node's** `next` member to point to the node that was previously at the beginning of the list
- The **second** step is to make `first` point to the new node
- **Practice: please revise the figure after these two statements were executed**

```
…
new_node->next = first;
first = new_node;
```

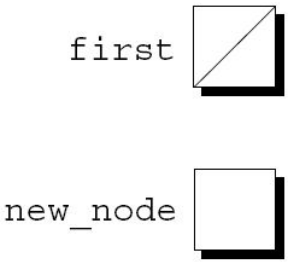


Inserting a Node at the Beginning of a Linked List

- These statements work even if the list is **empty**.
- Let's **trace the process** of inserting two nodes into an empty list.
- We'll **insert** a node containing the number **10** first, **followed** by a node containing **20**.

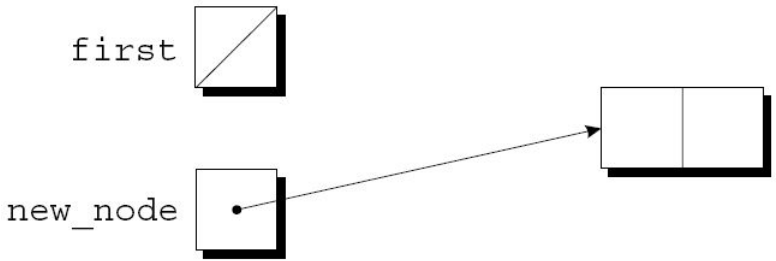
Inserting a Node at the Beginning of a Linked List

```
…
struct node *new_node, *first;
first = NULL;
new_node = malloc(sizeof(struct node));
new_node->value = 10;
new_node->next = first;
first = new_node;
new_node = malloc(sizeof(struct node))
new_node->value = 20;
new_node->next = first;
first = new_node;
…
```



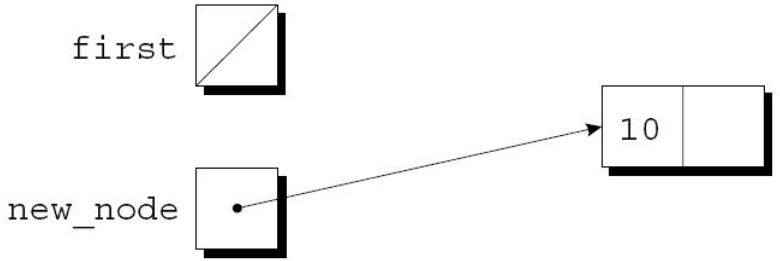
Inserting a Node at the Beginning of a Linked List

```
…
struct node *new_node, *first;
first = NULL;
new_node = malloc(sizeof(struct node));
new_node->value = 10;
new_node->next = first;
first = new_node;
new_node = malloc(sizeof(struct node))
new_node->value = 20;
new_node->next = first;
first = new_node;
…
```



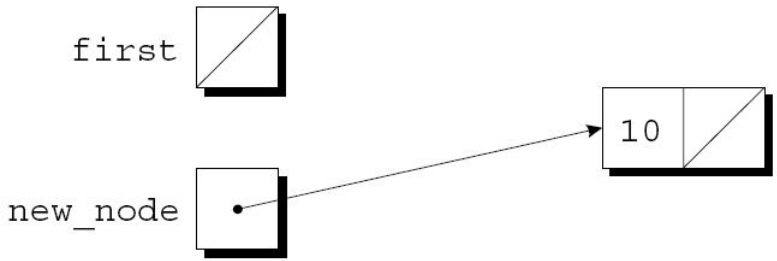
Inserting a Node at the Beginning of a Linked List

```
…
struct node *new_node, *first;
first = NULL;
new_node = malloc(sizeof(struct node));
new_node->value = 10;
new_node->next = first;
first = new_node;
new_node = malloc(sizeof(struct node))
new_node->value = 20;
new_node->next = first;
first = new_node;
…
```



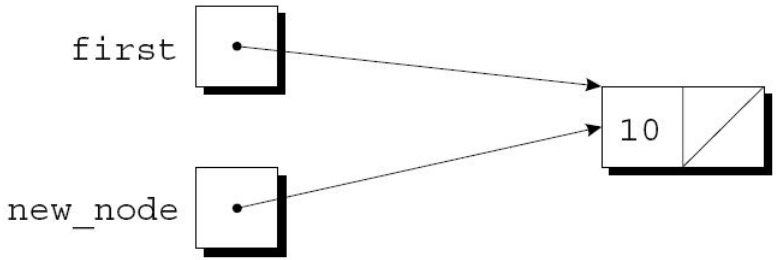
Inserting a Node at the Beginning of a Linked List

```
…
struct node *new_node, *first;
first = NULL;
new_node = malloc(sizeof(struct node));
new_node->value = 10;
new_node->next = first;
first = new_node;
new_node = malloc(sizeof(struct node))
new_node->value = 20;
new_node->next = first;
first = new_node;
…
```



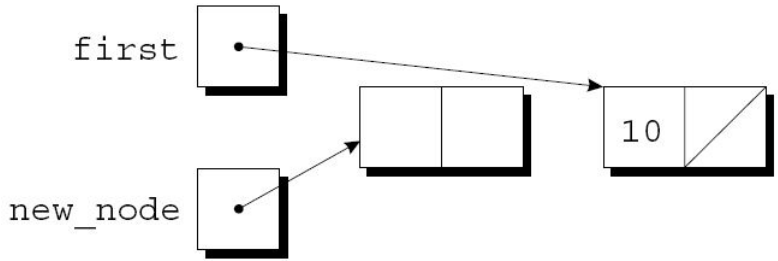
Inserting a Node at the Beginning of a Linked List

```
…
struct node *new_node, *first;
first = NULL;
new_node = malloc(sizeof(struct node));
new_node->value = 10;
new_node->next = first;
first = new_node;
new_node = malloc(sizeof(struct node))
new_node->value = 20;
new_node->next = first;
first = new_node;
…
```



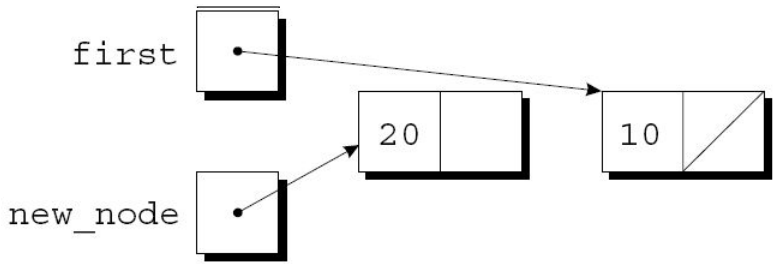
Inserting a Node at the Beginning of a Linked List

```
…
struct node *new_node, *first;
first = NULL;
new_node = malloc(sizeof(struct node));
new_node->value = 10;
new_node->next = first;
first = new_node;
new_node = malloc(sizeof(struct node))
new_node->value = 20;
new_node->next = first;
first = new_node;
…
```



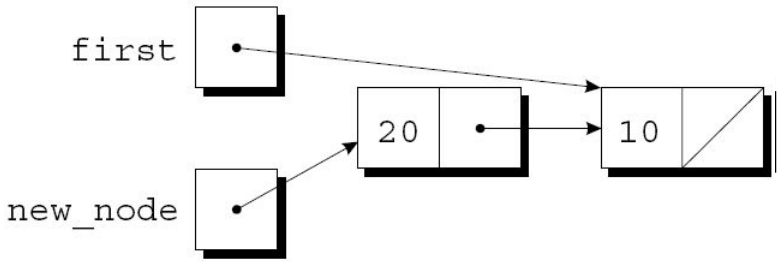
Inserting a Node at the Beginning of a Linked List

```
…
struct node *new_node, *first;
first = NULL;
new_node = malloc(sizeof(struct node));
new_node->value = 10;
new_node->next = first;
first = new_node;
new_node = malloc(sizeof(struct node))
new_node->value = 20;
new_node->next = first;
first = new_node;
…
```



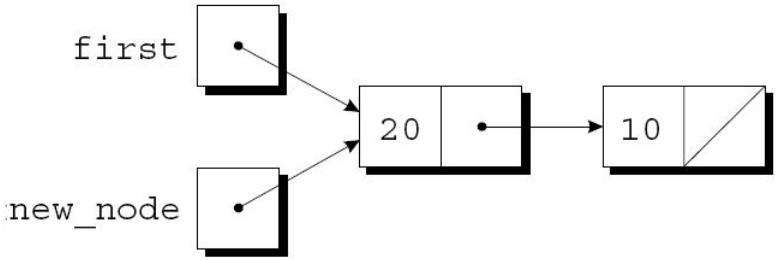
Inserting a Node at the Beginning of a Linked List

```
…
struct node *new_node, *first;
first = NULL;
new_node = malloc(sizeof(struct node));
new_node->value = 10;
new_node->next = first;
first = new_node;
new_node = malloc(sizeof(struct node))
new_node->value = 20;
new_node->next = first;
first = new_node;
…
```



Inserting a Node at the Beginning of a Linked List

```
…
struct node *new_node, *first;
first = NULL;
new_node = malloc(sizeof(struct node));
new_node->value = 10;
new_node->next = first;
first = new_node;
new_node = malloc(sizeof(struct node))
new_node->value = 20;
new_node->next = first;
first = new_node;
…
```



Inserting a Node at the Beginning of a Linked List

- A that inserts a node containing `n` into a linked list, which pointed to by `list`

```
struct node *add_to_list(struct node *list, int n){
      struct node *new_node;
      new_node = malloc(sizeof(struct node));
      if (new_node == NULL) {
        printf("Error: malloc failed in add_to_list\n");
        exit();
      }
      new_node->value = n;
      new_node->next = list;
      return new_node;
}
```

Please track the process this function by drawing the node figures

Inserting a Node at the Beginning of a Linked List

- Note that `add_to_list` returns a pointer to the newly created node (now at the beginning of the list).

```
struct node *add_to_list(struct node *list, int n){
      struct node *new_node;
      new_node = malloc(sizeof(struct node));
      if (new_node == NULL) {
        printf("Error: malloc failed in add_to_list\n");
        exit();
      }
      new_node->value = n;
      new_node->next = list;
      return new_node;
}
```

Inserting a Node at the Beginning of a Linked List

- When we call `add_to_list`, we'll need to store its return value into `first`
- Getting `add_to_list` to **update** `first` directly

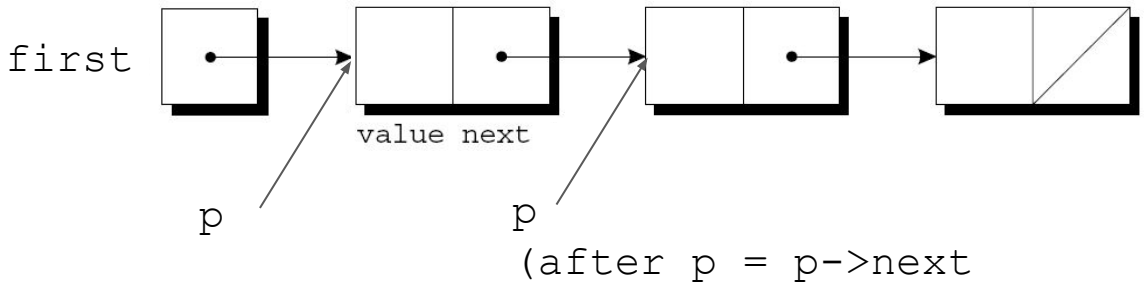
```
…
first = add_to_list(first, 10);
first = add_to_list(first, 20);
```

Let’s Take a Break

Searching a Linked List

- Although a `while` loop can be used to search a list, the `for` statement is often better.
- A loop that visits the nodes in a linked list, using a **pointer variable** `p` to keep track of the "current" node
- A loop of this form can be used in a function that searches a list for an integer `n`.

```
for (p = first; p != NULL; p = p->next)
```



Searching a Linked List

- If it finds `n`, the function will **return** a **pointer to the node** containing `n`; otherwise, it will return a null pointer.

```
struct node *search_list(struct node *list, int n){
      struct node *p;

      for (p = list; p != NULL; p = p->next){
        if (p->value == n){
          return p;
        }
      }
      return NULL;
}
```

Searching a Linked List

- There are many other ways to write `search_list`.
- One alternative is to eliminate the `p` variable, instead using `list` itself to keep track of the current node

```
struct node *search_list(struct node *list, int n){
      for (; list != NULL; list = list->next){
        if (list->value == n){
          return list;
        }
      }
      return NULL;
}
```

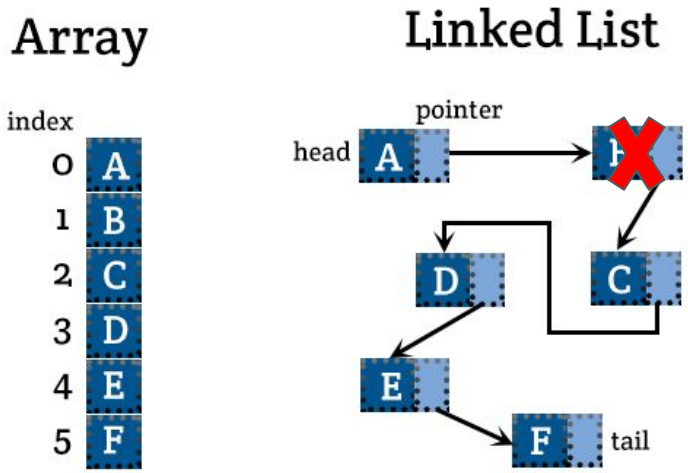
Searching a Linked List

- Another alternative can be the following:
- Since list is `NULL` if we reach the end of the list, returning `list` is correct even if we don't find `n`.

```
struct node *search_list(struct node *list, int n){
      while (list != NULL && list->value != n){
        list = list->next;
       }
      return list;
}
```

Deleting a Node from a Linked List

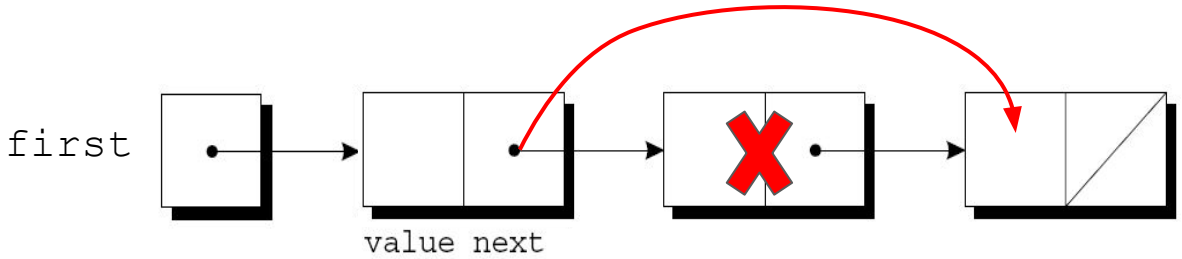
- A big advantage of storing data in a linked list is that we can **easily delete nodes**.
- We cannot delete (free memeory of) an element from an array.



ref. Open4Tech

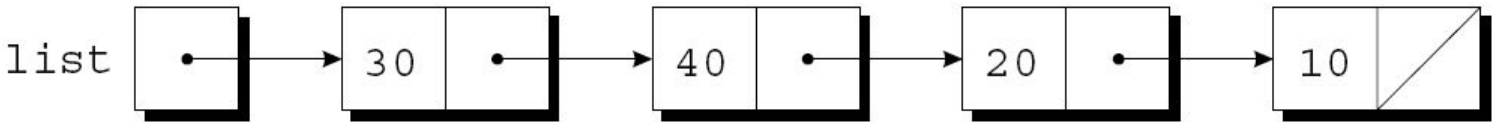
Deleting a Node from a Linked List

- Deleting a node involves three steps:
  1. Locate the node to be deleted.
  2. Alter the previous node so that it "bypasses" the deleted node.
  3. Call `free` to reclaim the space occupied by the deleted node.
- Step 1 is harder than it looks, because step 2 requires changing the *previous* node.



Deleting a Node from a Linked List

- One solution is keeping a **pointer** to the previous node (`prev`) and the other pointer to the current node (`cur`).
- Assume that `list` points to the list to be searched and `n` is the integer to be deleted.
  - Assuming that `n` is 20

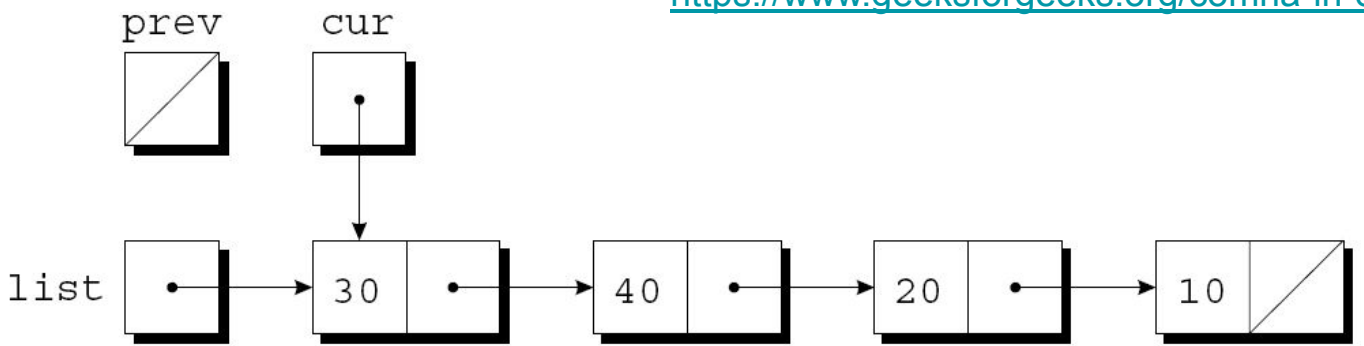


Deleting a Node from a Linked List

- A loop that implements step 1 (Locate the node to be deleted)

```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next);
```

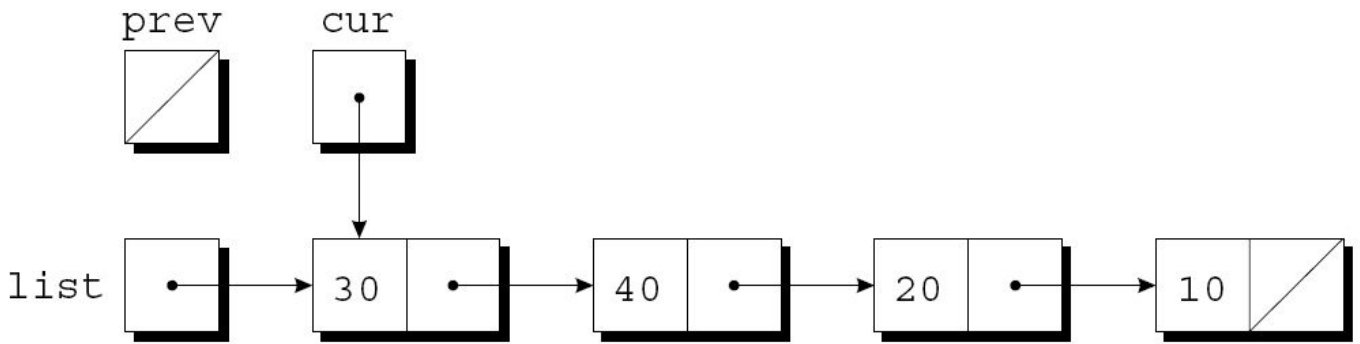
https://www.geeksforgeeks.org/comna-in-c-and-c/



Deleting a Node from a Linked List

- The test is true, since `cur` is pointing to a node and the node doesn't contain 20.

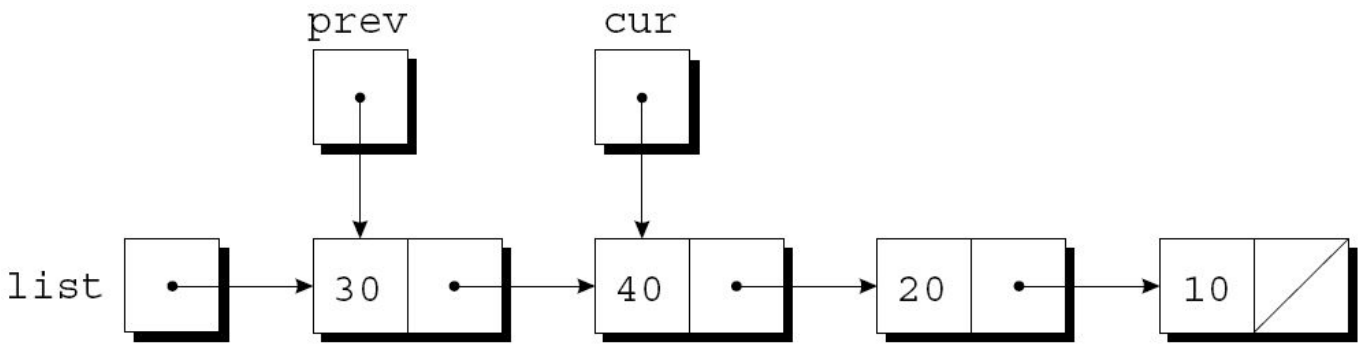
```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next);
```



Deleting a Node from a Linked List

- After `prev = cur, cur = cur->next` has been executed

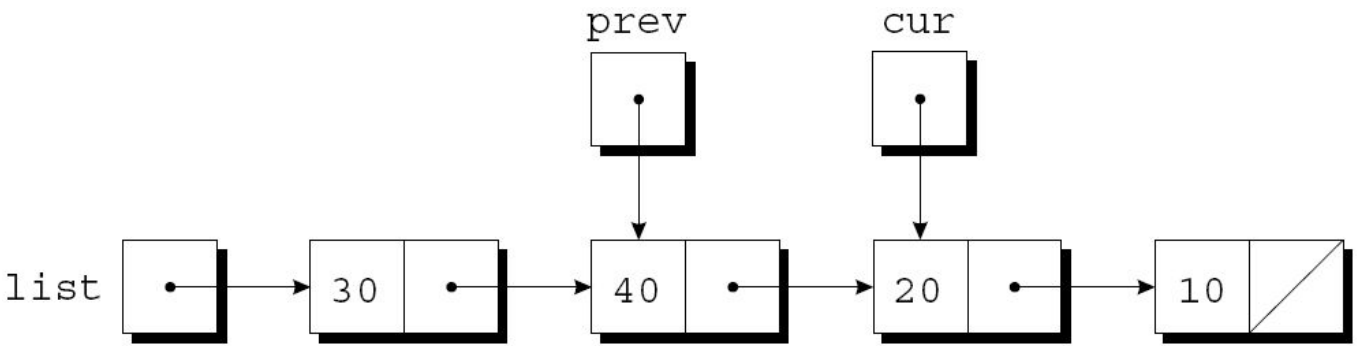
```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next);
```



Deleting a Node from a Linked List

- The test is again true, so `prev = cur, cur = cur->next` is executed again

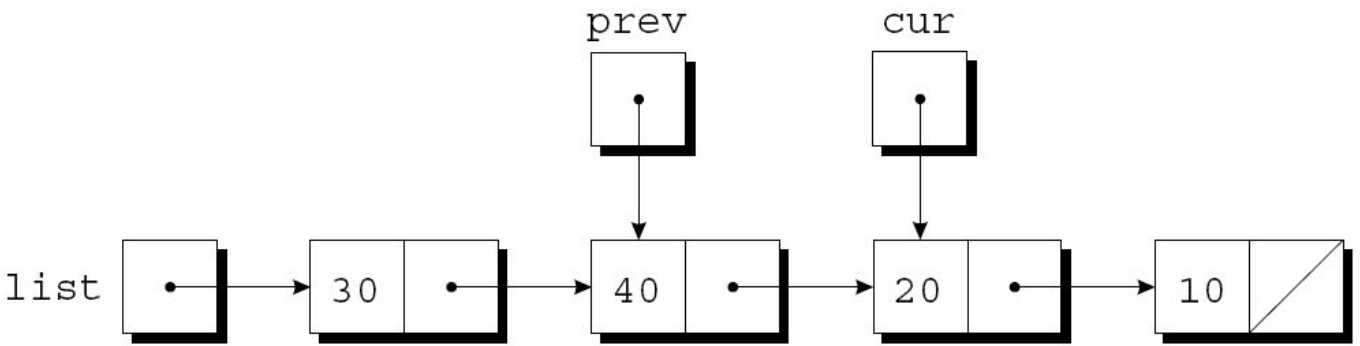
```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next);
```



Deleting a Node from a Linked List

- Since `cur` now points to the node containing 20, the test is false
- the loop terminates.

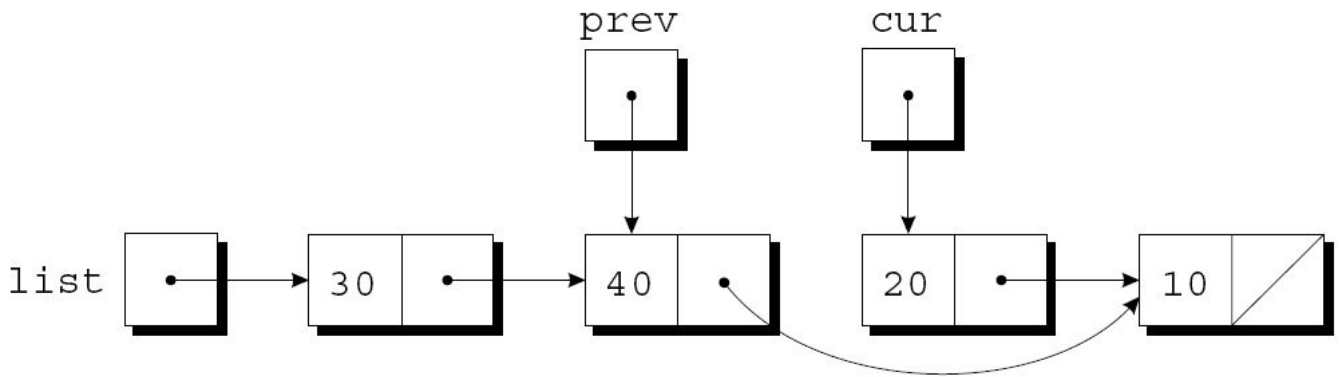
```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next);
```



Deleting a Node from a Linked List

- Next, we'll perform the step 2 (previous node bypasses deleted node) by the statement.

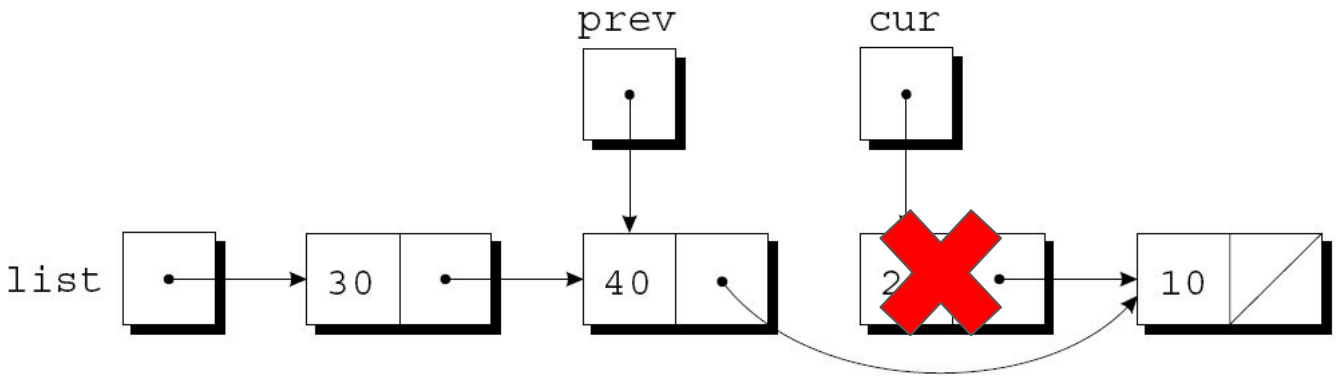
```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next);
prev->next = cur->next;
```



Deleting a Node from a Linked List

- Step 3 is to release the memory occupied by the current node:

```
for (cur = list, prev = NULL;
     cur != NULL && cur->value != n;
     prev = cur, cur = cur->next);
prev->next = cur->next;
free(cur);
```



```
struct node *delete_from_list(struct node *list, int n){
  struct node *cur, *prev;

  for (cur = list, prev = N
       cur != NULL && cur->value != n;
       prev = cur, cur = cur->next);
  if (cur == NULL){
    return list;                /* n was not found */
  }
  if (prev == NULL){
    list = list->next;          /* n is in the first node */
  }
  else{
    prev->next = cur->next;     /* n is in some other node */
  }
  free(cur);
  return list;
}
```

When given a list and an integer `n`, the function deletes the first node containing `n`.

```
struct node *delete_from_list(struct node *list, int n){
  struct node *cur, *prev;

  for (cur = list, prev = NULL;
       cur != NULL && cur->value != n;
       prev = cur, cur = cur->next);
  if (cur == NULL){
    return list;                  /* n was not found */
  }
  if (prev == NULL){
    list = list->next;            /* n is in the first node */
  }
  else{
    prev->next = cur->next;       /* n is in some other node */
  }
  free(cur);
  return list;
}
```

If no node contains `n`, does nothing.

```
struct node *delete_from_list(struct node *list, int n){
  struct node *cur, *prev;

  for (cur = list, prev = NULL;
       cur != NULL && cur->value != n;
       prev = cur, cur = cur->next);
  if (cur == NULL){
    return list;                /* n was not found */
  }
  if (prev == NULL){
    list = list->next;          /* n is in the first node */
  }
  else{
    prev->next = cur->nex
  }
  free(cur);
  return list;
}
```

Deleting the first node in the list is a special case that requires a different bypass step.

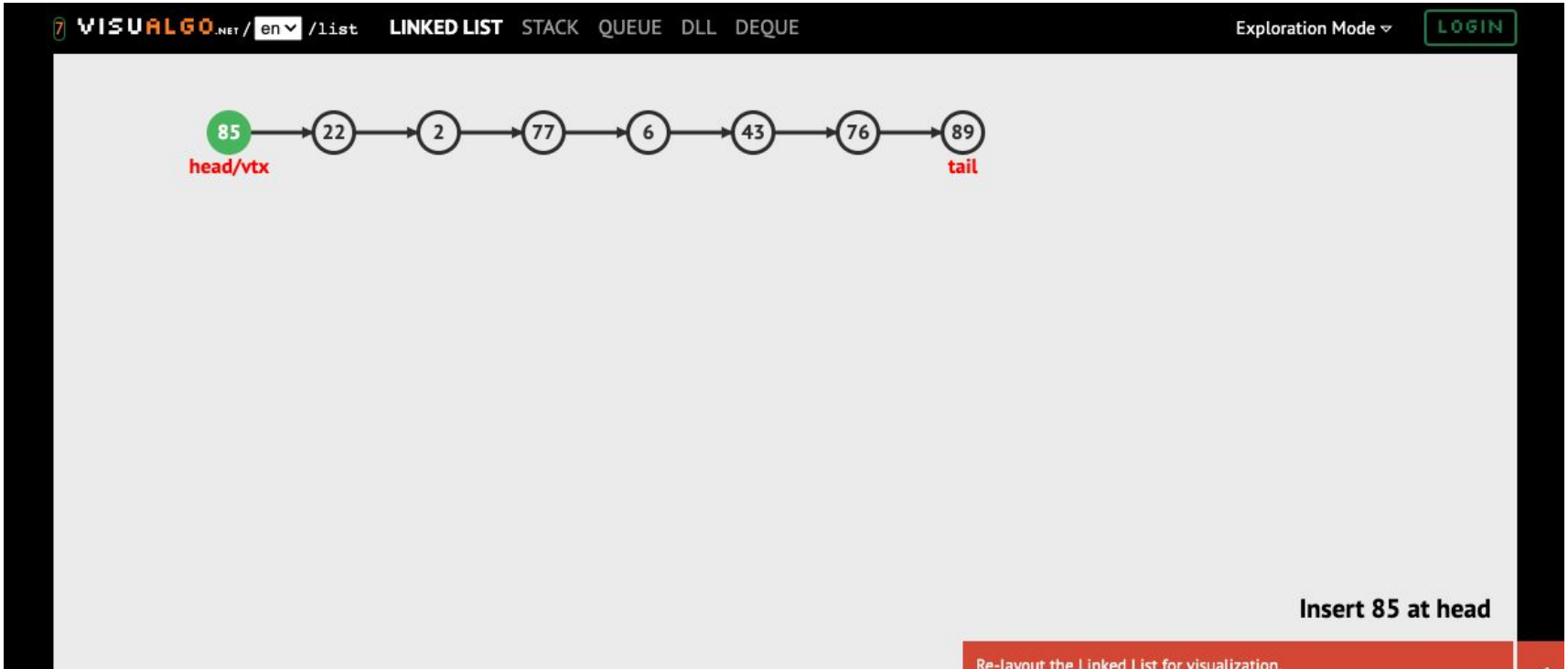
```
struct node *delete_from_list(struct node *list, int n){
  struct node *cur, *prev;

  for (cur = list, prev = NULL;
       cur != NULL && cur->value != n;
       prev = cur, cur = cur->next);
  if (cur == NULL){
    return list;                /* n was not found */
  }
  if (prev == NULL){
    list = list->next;          /* n is in the first node */
  }
  else{
    prev->next = cur->next;     /* n is in some other node */
  }
  free(cur);
  return list;
}
```

If found `n` in other node, do the normal bypass step.

Extra Reading/Playing

- https://visualgo.net/en/list



Summary

- Introduce Linked Lists
  - Declaring a Node Type
  - creating a node
- Common Operations of Linked Lists
  - inserting a node at the beginning of a (linked) list
  - searching for a node
  - deleting a node