Linked Lists

Marcus Birkenkrahe

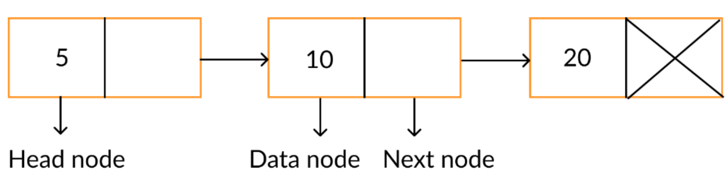
December 2, 2024

README

- This lecture with practice exercises contains an introduction to the *linked list* data structure.
- Covered are only:
  1. Definition and use of linked lists
  2. Declaration and initialization of a list
  3. Selecting list member
  4. Inserting a node at the beginning of a linked list
- Not covered:
  1. Searching in a linked list
  2. Deleting a node from a linked list
  3. Doubly linked lists (pointers go both directions)

Linked lists

- A **linked list** consists of a chain of structures called **nodes**. Each node contains a pointer to the next node in the chain. The last node contains a null pointer:



- A linked list is more flexible than an array to store a collection of data items: Nodes can easily be inserted and deleted to grow and shrink the list.
- Arrays have "random access": Any of its elements can be accessed in the same amount of time using array index subscripting or pointer arithmetic.
- Accessing a node in a linked list is fast if the node is close to the beginning of the list, and slow if it's near the end.
- A list is an ADT, an Abstract Data Type: We need to define common operations on linked lists: inserting a node at the beginning of the list, searching for a node, and deleting a node.

Declaring a Node Type

- We need a structure that represents a single node in a linked list.
- Example: A simple `node` structure with one integer member and a pointer member `next` to the next `node`.

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

- `node` is self-referential because it contains a `node` structure.
- The tag `node` is not special or reserved, it could be any name.
- Normally, you can create a named structure either with a name tag like here, or with a `typedef` name. When a structure is self-referential, with a member that points to the same type of structure, you must use a structure tag.
- Exemplary use of `node`: This program declares a `node` called `first`. Setting it to `NULL` means that the list is initially empty.

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

```
}; // node with one integer member ‘value‘


struct node *first = NULL;
// printf("%d\n", first->value);  // segmentation fault!
```

- There is nothing to see here: `first` does not point to a valid memory location, and attempting to access `first->value` results in undefined behavior.

Creating a Node

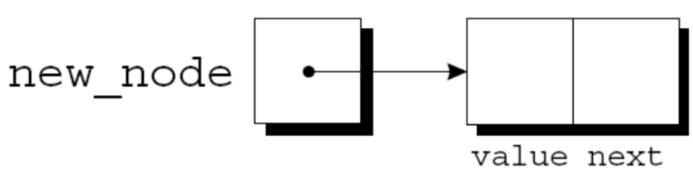
- Constructing a list means creating nodes one by one:

  1. Allocate memory for a new node.
  2. Store data in the node.
  3. Insert the node into the list.

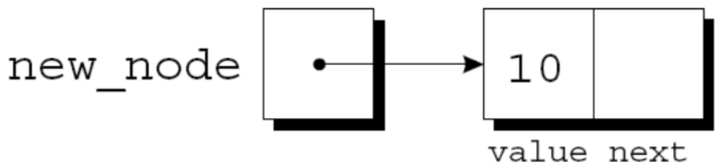
- Code:

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

  // allocate memory for ‘new_node‘
struct node *new_node = malloc(sizeof(struct node));

// store data in the ‘value‘ member of ‘new_node‘
(*new_node).value = 10;
```

- After the first command, `new_node` now points to a block of memory just large enough to hold a `node` structure with its members.



- After the second command, dereferencing `new_node` allows us to access its `value` member.



- We can now print the `value` of the `new_node`:

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

  // allocate memory for `new_node`
struct node *new_node = malloc(sizeof(struct node));

// store data in the `value` member of `new_node`
(*new_node).value = 10;

printf("New node value: %d\n",(*new_node).value);
```

```
New node value: 10
```

- The command must be formatted `(*new_node).value = 10;` rather than `*new_node.value = 10;` because the dot-operator otherwise takes precedence over the indirection operator.

The right arrow selection operator ->

- Accessing a member of a structure using a pointer is so common that C provides a special operator for it, the "right arrow selection" `->`
- Using the arrow operator instead of indirection + selection with `(*new_node).value`:

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

  // allocate memory for `new_node`
struct node *new_node = malloc(sizeof(struct node));

// store data in the `value` member of `new_node`
(*new_node).value = 10;

printf("New node value: %d\n",new_node->value);

new_node->value = 11;

printf("New node value: %d\n",new_node->value);
```

```
New node value: 10
New node value: 11
```

- The `->` operator produces an *lvalue*, which is why it can be used in the `printf` call or in an assignment with `=`.
- You can use `->` for all members of a structure, and you don't have to remember which ones are pointers. This is especially useful in linked lists with pointers like `*next` to structures that contain other pointers.

Practice: Input with right arrow selection using `scanf`

- Input the `value` 12 using `scanf`.
- Solution (with the structure definition)

  1. Generate an input file:

     ```
     echo 12 > nodeInput
     cat nodeInput
     ```

     ```
     12
     ```

2. Using the input file

```
// structure definition
struct node {
  int value;  // this is node->value
  struct node *next; // this is node->next
};

// new node definition
struct node *new_node = malloc(sizeof(struct node));

// get input for new_node->value
scanf("%d\n", &new_node->value);

// print output
printf("%d\n", new_node->value);
```

```
12
```

- Notice that `scanf` requires the *address-of* `new_node->value` even though `new_node` is a pointer. But `new_node->value` is an `int` so we need to convert it to an address for `scanf`.

Practice: Input with right arrow selection using `main(int argc,char **argv)`

- Create another solution with a complete (not `void`) `main` function:

  1. Tangle the file `newNode.c`
  2. Test it on the command-line (in a `bash` block)

```
struct node {
  int value;
  struct node *next;
};

int main(int argc, char **argv)
{
  struct node *new_node = malloc(sizeof(struct node));
```

```
  if (argc > 1) {
    new_node->value = atoi(argv[1]); // convert char argument to integer
    printf("Value = %d\n", new_node->value);
  } else {
    printf("Usage: %s <number>\n", argv[0]);
    return 0;
  }
}
```

```
Usage: /tmp/babel-OCXUsf/C-bin-fUU7TW <number>
```

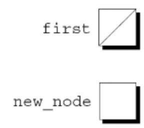
```
if [ -e "./newNode.c" ]; then
    gcc newNode.c -o newNode
    ./newNode
    ./newNode 12
else
    echo "File does not exist"
fi
```

```
File does not exist
```

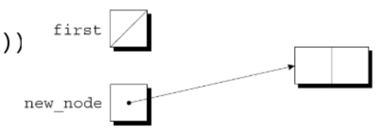
Inserting a Node at the Beginning of a Linked List

- You can add nodes at any point in the list: At the beginning, at the end, or anywhere in the middle. Adding a new element at the beginning is the easiest place to do this.
- It takes two statements to insert the node into the list:
  1. Make the new node's `next` member point to the node that was previously at the beginning of the list: `new_node->next = first;`
  2. Make the `first` node point to the new node: `first = new_node;`
- Illustration with a little more detail:
  1. Create a first (`NULL`) pointer and a `new_node`, then make a `node` list item with `data` and `next` members:

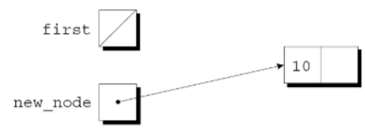
```
first = NULL;
```




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

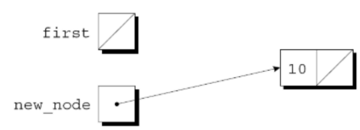



```
new_node->value = 10;
```

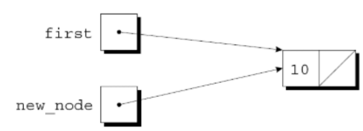


2. Point `first` pointer at the first list item. Now both `first` and `new_node` point at the same item. Then create a second `node`:

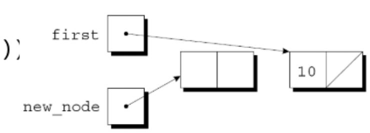
```
new_node->next = first;
```




```
first = new_node;
```

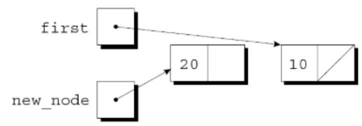



```
new_node =
  malloc(sizeof(struct node))
```

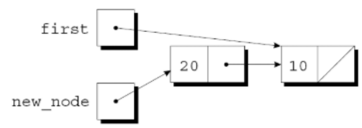


3. Create a second list item, make its `next` member point at the first list item, and then point `first` at the new item.

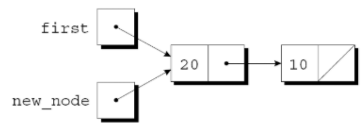
```
new_node->value = 20;
```




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




```
first = new_node;
```



- These statements work even if the list is empty.
- Example code:

```
struct node {
  int data;
  struct node *next;
};

// declare two pointers
struct node *first = NULL;
struct node *new_node;

// first list item
new_node = malloc(sizeof(struct node));
new_node->data = 10;
new_node->next = first;
first = new_node;

// second list item
new_node = malloc(sizeof(struct node));
new_node->data = 20;
new_node->next = first;
first = new_node;
```

- Print the list so far:

```
struct node {
```

```
  int data;
  struct node *next;
};

// declare two pointers
struct node *first = NULL;
struct node *new_node;

// first list item
new_node = malloc(sizeof(struct node));
new_node->data = 10;
new_node->next = first;
first = new_node;

// second list item
new_node = malloc(sizeof(struct node));
new_node->data = 20;
new_node->next = first;
first = new_node;

  // print the list so far
struct node *item = first;
while (item != NULL) {
  printf("item is at %p; next is at %p; data is %d\n",
     item, item->next, item->data);
  item = item->next;
 }
```

```
item is at 0x5b71574cb2c0; next is at 0x5b71574cb2a0; data is 20
item is at 0x5b71574cb2a0; next is at (nil); data is 10
```

Practice: Modify the code to add a third list member

Add the code chunk for two list members, then:

1. Create a new node.
2. Store 30 in the new node.
3. Point `next` at the previously first member.

4. Point `first` at the new member.
5. Print the list so far.

```
struct node {
  int data;
  struct node *next;
};

// declare two pointers
struct node *first = NULL;
struct node *new_node;

// first list item
new_node = malloc(sizeof(struct node));
new_node->data = 10;
new_node->next = first;
first = new_node;

// second list item
new_node = malloc(sizeof(struct node));
new_node->data = 20;
new_node->next = first;
first = new_node;

  // third list item
new_node = malloc(sizeof(struct node)); // #1
new_node->data = 30; // #2
new_node->next = first; // #3
first = new_node;

struct node *item = first; // start at the beginning
while (item != NULL) {
  printf("item is at %p; next is at %p; data is %d\n",
       item, item->next, item->data);
  item = item->next;
 }
```

```
item is at 0x561cc94ee2e0; next is at 0x561cc94ee2c0; data is 30
item is at 0x561cc94ee2c0; next is at 0x561cc94ee2a0; data is 20
item is at 0x561cc94ee2a0; next is at (nil); data is 10
```

Practice: Print list with a function `print_list`

- Write a function `print_list` to print the list.

```
struct node {
  int data;
  struct node *next;
};

// print list
// return: nothing
// params: pointer to list node structure
void print_list(struct node *list);

int main(void)
{

  // declare two pointers
  struct node *first = NULL;
  struct node *new_node;

  // first list item
  new_node = malloc(sizeof(struct node));
  new_node->data = 10;
  new_node->next = first;
  first = new_node;

  // second list item
  new_node = malloc(sizeof(struct node));
  new_node->data = 20;
  new_node->next = first;
  first = new_node;

  print_list(first);
  return 0;
}

void print_list(struct node *first)
{
  struct node *item = first;
```

```
  while (item != NULL) {
    printf("item is at %p; next is at %p; data is %d\n",
     (void *)item, (void *)item->next, item->data);
    item = item->next;
  }
}
```

```
item is at 0x55a792e2c2c0; next is at 0x55a792e2c2a0; data is 20
item is at 0x55a792e2c2a0; next is at (nil); data is 10
```

- Making the cast explicit in `printf` ensures portability (treat pointer as generic) & avoids implicit conversion warnings.

Practice: Create an insertion function `add_to_list`

- Write a function `add_to_list` that inserts a node into a linked list.

```
struct node {
  int data;
  struct node *next;
};

// task: print list
// return: nothing
// params: pointer to list node structure
void print_list(struct node *list);

// task: add to list from beginning
// return: pointer to new node (now beginning of list)
// params: pointer to list, data to store
struct node *add_to_list(struct node *list, int n);

int main(void)
{

  // declare two pointers
  struct node *first = NULL;
```

```
  for (int i=1; i<4; i++)
    first = add_to_list(first,i*10);
  print_list(first);
  return 0;
}

void print_list(struct node *first)
{
  struct node *item = first;
  while (item != NULL) {
    printf("item is at %p; next is at %p; data is %d\n",
      (void *)item, (void *)item->next, item->data);
    item = item->next;
  }
}

struct node *add_to_list(struct node *first, int n)
{
  struct node *new_node; // declare new node
  new_node = malloc(sizeof(struct node)); // allocate new member
  new_node->data = n; // store data member
  new_node->next = first; // repoint next member to previous member
  first = new_node; // repoint beginning of list to new member
  return new_node;
}
```

```
item is at 0x5686761392e0; next is at 0x5686761392c0; data is 30
item is at 0x5686761392c0; next is at 0x5686761392a0; data is 20
item is at 0x5686761392a0; next is at (nil); data is 10
```

- When the new node is a `NULL` pointer, no memory should be added, and it is better to add this check after the allocation of `new_node`:

```
if (new_node == NULL) {
  printf("Error: malloc failed in add_to_list\n");
  exit (EXIT_FAILURE);
}
```