Linked lists

A doubly linked list

An example of code showing how nodes can be inserted at a doubly linked list, how the list can easily be reversed, and how it can be printed in reverse.

```c
#include <stdio.h>
#include <stdlib.h>

/* This data is not always stored in a structure, but it is sometimes for ease of us
struct Node {
  /* Sometimes a key is also stored and used in the functions */
  int data;
  struct Node* next;
  struct Node* previous;
};

void insert_at_beginning(struct Node **pheadNode, int value);
void insert_at_end(struct Node **pheadNode, int value);

void print_list(struct Node *headNode);
void print_list_backwards(struct Node *headNode);

void free_list(struct Node *headNode);
```

```
  /* Sometimes in a doubly linked list the last node is also stored */
  struct Node *head = NULL;

  printf("Insert a node at the beginning of the list.\n");
  insert_at_beginning(&head, 5);
  print_list(head);

  printf("Insert a node at the beginning, and then print the list backwards\n");
  insert_at_beginning(&head, 10);
  print_list_backwards(head);

  printf("Insert a node at the end, and then print the list forwards.\n");

  insert_at_end(&head, 15);
  print_list(head);

  free_list(head);

  return 0;
}

void print_list_backwards(struct Node *headNode) {
  if (NULL == headNode)
  {
    return;
  }
  /*
  Iterate through the list, and once we get to the end, iterate backwards to print
  out the items in reverse order (this is done with the pointer to the previous node
  This can be done even more easily if a pointer to the last node is stored.
```

```
    while (i->next != NULL) {
      i = i->next; /* Move to the end of the list */
    }

    while (i != NULL) {
      printf("Value: %d\n", i->data);
      i = i->previous;
    }
  }

  void print_list(struct Node *headNode) {
    /* Iterate through the list and print out the data member of each node */
    struct Node *i;
    for (i = headNode; i != NULL; i = i->next) {
      printf("Value: %d\n", i->data);
    }
  }

  void insert_at_beginning(struct Node **pheadNode, int value) {
    struct Node *currentNode;

    if (NULL == pheadNode)
    {
      return;
    }
    /*
    This is done similarly to how we insert a node at the beginning of a singly linked
    list, instead we set the previous member of the structure as well
    */
    currentNode = malloc(sizeof *currentNode);
```

```
  currentNode->data = value;

  if (*pheadNode == NULL) { /* The list is empty */
    *pheadNode = currentNode;
    return;
  }

  currentNode->next = *pheadNode;
  (*pheadNode)->previous = currentNode;
  *pheadNode = currentNode;
}

void insert_at_end(struct Node **pheadNode, int value) {
  struct Node *currentNode;

  if (NULL == pheadNode)
  {
    return;
  }

  /*
  This can, again be done easily by being able to have the previous element.  It
  would also be even more useful to have a pointer to the last node, which is commor
  used.
  */

  currentNode = malloc(sizeof *currentNode);
  struct Node *i = *pheadNode;
```

```
  currentNode->previous = NULL;

  if (*pheadNode == NULL) {
    *pheadNode = currentNode;
    return;
  }

  while (i->next != NULL) { /* Go to the end of the list */
    i = i->next;
  }

  i->next = currentNode;
  currentNode->previous = i;
}

void free_list(struct Node *node) {
  while (node != NULL) {
    struct Node *next = node->next;
    free(node);
    node = next;
  }
}
```

Note that sometimes, storing a pointer to the last node is useful (it is more efficient to simply be able to jump straight to the end of the list than to need to iterate through to the end):

In which case, updating it upon changes to the list is needed.

Sometimes, a key is also used to identify elements. It is simply a member of the Node structure:

```c
struct Node {
  int data;
  int key;
  struct Node* next;
  struct Node* previous;
};
```

The key is then used when any tasks are performed on a specific element, like deleting elements.

Inserting a node at the beginning of a singly linked list

The code below will prompt for numbers and continue to add them to the beginning of a linked list.

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data;
  struct Node* next;
};


void insert_node (struct Node **head, int nodeValue);
void print_list (struct Node *head);

int main(int argc, char *argv[]) {
  struct Node* headNode;
  headNode = NULL; /* Initialize our first node pointer to be NULL. */
  size_t listSize, i;
  do {
    printf("How many numbers would you like to input?\n");
  } while(1 != scanf("%zu", &listSize));

  for (i = 0; i < listSize; i++) {
    int numToAdd;
    do {
      printf("Enter a number:\n");
    } while (1 != scanf("%d", &numToAdd));

    insert_node (&headNode, numToAdd);
    printf("Current list after your inserted node: \n");
    print_list(headNode);
```

```
}

void print_list (struct Node *head) {
  struct node* currentNode = head;

  /* Iterate through each link. */
  while (currentNode != NULL) {
      printf("Value: %d\n", currentNode->data);
      currentNode = currentNode -> next;
  }
}

void insert_node (struct Node **head, int nodeValue) {
  struct Node *currentNode = malloc(sizeof *currentNode);
  currentNode->data = nodeValue;
  currentNode->next = (*head);

  *head = currentNode;
}
```

Explanation for the Insertion of Nodes

In order to understand how we add nodes at the beginning, let's take a look at possible scenarios:

```
| HEAD | --> NULL
```

The line `currentNode->next = *headNode;` will assign the value of `currentNode->next` to be `NULL` since `headNode` originally starts out at a value of `NULL` .

Now, we want to set our head node pointer to point to our current node.

```
-----      -------------
|HEAD | --> |CURRENTNODE| --> NULL /* The head node points to the current node */
 -----      -------------
```

This is done with `*headNode = currentNode;`

1. The list is already populated; we need to add a new node to the beginning. For the sake of simplicity, let's start out with 1 node:

```
-----    -----------
 HEAD --> FIRST NODE --> NULL
```

With `currentNode->next = *headNode`, the data structure looks like this:

```
---------        -----    ---------------------
currentNode -->  HEAD --> POINTER TO FIRST NODE --> NULL
---------        -----    ---------------------
```

Which, obviously needs to be altered since `*headNode` should point to `currentNode`.

```
----    -----------    ---------------
HEAD -> currentNode -->     NODE       -> NULL
----    -----------    ---------------
```

This is done with `*headNode = currentNode;`

Inserting a node at the nth position

So far, we have looked at **inserting a node at the beginning of a singly linked list**. However, most of the times you will want to be able to insert nodes elsewhere as well. The

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data;
  struct Node* next;
};

struct Node* insert(struct Node* head, int value, size_t position);
void print_list (struct Node* head);

int main(int argc, char *argv[]) {
  struct Node *head = NULL; /* Initialize the list to be empty */

  /* Insert nodes at positions with values: */
  head = insert(head, 1, 0);
  head = insert(head, 100, 1);
  head = insert(head, 21, 2);
  head = insert(head, 2, 3);
  head = insert(head, 5, 4);
  head = insert(head, 42, 2);

  print_list(head);
  return 0;
}

struct Node* insert(struct Node* head, int value, size_t position) {
  size_t i = 0;
```

```
/* Create our node */
currentNode = malloc(sizeof *currentNode);
/* Check for success of malloc() here! */

/* Assign data */
currentNode->data = value;

/* Holds a pointer to the 'next' field that we have to link to the new node.
   By initializing it to &head we handle the case of insertion at the beginning. */
struct Node **nextForPosition = &head;
/* Iterate to get the 'next' field we are looking for.
   Note: Insert at the end if position is larger than current number of elements.
for (i = 0; i < position && *nextForPosition != NULL; i++) {
    /* nextForPosition is pointing to the 'next' field of the node.
       So *nextForPosition is a pointer to the next node.
       Update it with a pointer to the 'next' field of the next node. */
    nextForPosition = &(*nextForPosition)->next;
}

/* Here, we are taking the link to the next node (the one our newly inserted node
point to) by dereferencing nextForPosition, which points to the 'next' field of th
that is in the position we want to insert our node at.
We assign this link to our next value. */
currentNode->next = *nextForPosition;

/* Now, we want to correct the link of the node before the position of our
new node: it will be changed to be a pointer to our new node. */
*nextForPosition = currentNode;

return head;
```

```
void print_list (struct Node *head) {
  /* Go through the list of nodes and print out the data in each node */
  struct Node* i = head;
  while (i != NULL) {
    printf("%d\n", i->data);
    i = i->next;
  }
}
```

Reversing a linked list

You can also perform this task recursively, but I have chosen in this example to use an iterative approach. This task is useful if you are **inserting all of your nodes at the beginning of a linked list** . Here is an example:

```c
#include <stdio.h>
#include <stdlib.h>

#define NUM_ITEMS 10

struct Node {
  int data;
  struct Node *next;
};

void insert_node(struct Node **headNode, int nodeValue, int position);
```

```
int main(void) {
  int i;
  struct Node *head = NULL;

  for(i = 1; i <= NUM_ITEMS; i++) {
    insert_node(&head, i, i);
  }
  print_list(head);

  printf("I will now reverse the linked list\n");
  reverse_list(&head);
  print_list(head);
  return 0;
}

void print_list(struct Node *headNode) {
  struct Node *iterator;

  for(iterator = headNode; iterator != NULL; iterator = iterator->next) {
    printf("Value: %d\n", iterator->data);
  }
}

void insert_node(struct Node **headNode, int nodeValue, int position) {
  int i;
  struct Node *currentNode = (struct Node *)malloc(sizeof(struct Node));
  struct Node *nodeBeforePosition = *headNode;
```

```
    currentNode->next = *headNode;
    *headNode = currentNode;
    return;
  }

  for (i = 0; i < position - 2; i++) {
    nodeBeforePosition = nodeBeforePosition->next;
  }

  currentNode->next = nodeBeforePosition->next;
  nodeBeforePosition->next = currentNode;
}

void reverse_list(struct Node **headNode) {
  struct Node *iterator = *headNode;
  struct Node *previousNode = NULL;
  struct Node *nextNode = NULL;

  while (iterator != NULL) {
    nextNode = iterator->next;
    iterator->next = previousNode;
    previousNode = iterator;
    iterator = nextNode;
  }

  /* Iterator will be NULL by the end, so the last node will be stored in
  previousNode.  We will set the last node to be the headNode */
  *headNode = previousNode;
```

Explanation for the Reverse List Method

We start the `previousNode` out as `NULL`, since we know on the first iteration of the loop, if we are looking for the node before the first head node, it will be `NULL`. The first node will become the last node in the list, and the next variable should naturally be `NULL`.

Basically, the concept of reversing the linked list here is that we actually reverse the links themselves. Each node's next member will become the node before it, like so:

```
Head -> 1 -> 2 -> 3 -> 4 -> 5
```

Where each number represents a node. This list would become:

```
1 <- 2 <- 3 <- 4 <- 5 <- Head
```

Finally, the head should point to the 5th node instead, and each node should point to the node previous of it.

Node 1 should point to `NULL` since there was nothing before it. Node 2 should point to node 1, node 3 should point to node 2, et cetera.

the list since the link to it is gone.

The solution to this problem is to simply store the next element in a variable (`nextNode`) before changing the link.

Remarks

The C language does not define a linked list data structure. If you are using C and need a linked list, you either need to use a linked list from an existing library (such as GLib) or write your own linked list interface. This topic shows examples for linked lists and double linked lists that can be used as a starting point for writing your own linked lists.

Singly linked list

The list contains nodes which are composed of one link called next.

Data structure

```c
struct singly_node
{
  struct singly_node * next;
};
```

Doubly linked list

The list contains nodes which are composed of two links called previous and next. The links are normally referencing to a node with the same structure.

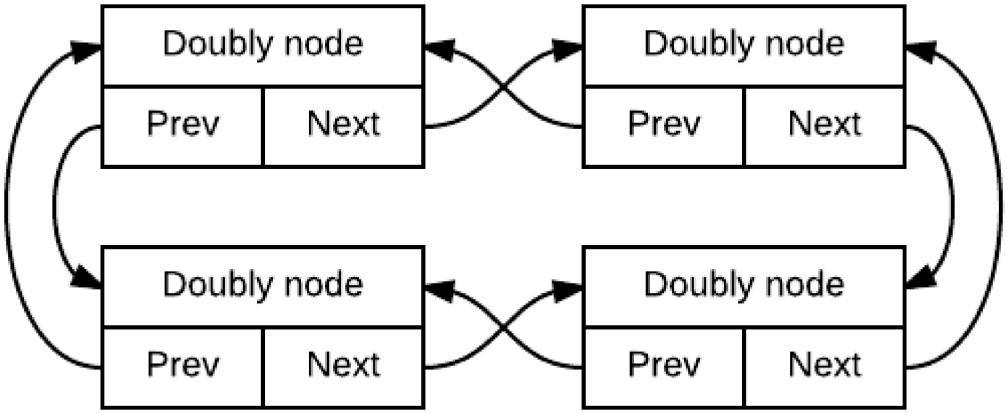
Data structure

```c
struct doubly_node
{
  struct doubly_node * prev;
  struct doubly_node * next;
};
```

Topoliges

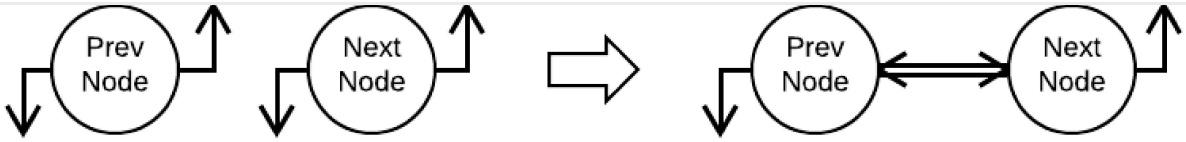
Linear or open

Circular or ring



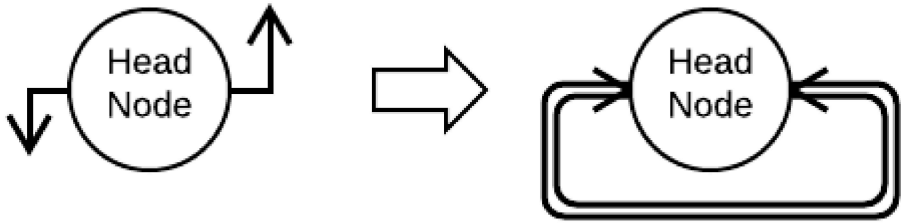
Procedures

Bind



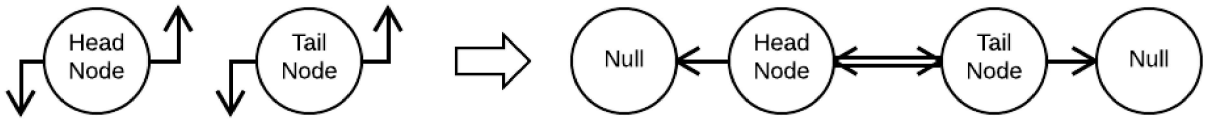

```c
void doubly_node_bind (struct doubly_node * prev, struct doubly_node * next)
{
  prev->next = next;
  next->prev = prev;
}
```

Making circularly linked list



```
  doubly_node_bind (head, head);
}
```

Making linearly linked list




```
void doubly_node_make_empty_linear_list (struct doubly_node * head, struct doubly_no
{
  head->prev = NULL;
  tail->next = NULL;
  doubly_node_bind (head, tail);
}
```

Insertion

```c
void doubly_node_insert_between
(struct doubly_node * prev, struct doubly_node * next, struct doubly_node * insertic
{
  doubly_node_bind (prev, insertion);
  doubly_node_bind (insertion, next);
}

void doubly_node_insert_before
(struct doubly_node * tail, struct doubly_node * insertion)
{
  doubly_node_insert_between (tail->prev, tail, insertion);
}

void doubly_node_insert_after
(struct doubly_node * head, struct doubly_node * insertion)
{
  doubly_node_insert_between (head, head->next, insertion);
}
```

Edit this page on GitHub