Liked list

Wednesday, September 6, 2017 10:22 AM

Example 4 (Traversing and Finding the number of elements in a linked list)

```
/*
Given a linked list head pointer, compute and return the number of nodes in
the list. BuildOneTwoThree() creates a linked list with 3 elements. Code is
not included
*/
int Length(struct node* head) {
      struct node* current = head;
      int count = 0;

      while (current != NULL) {
            count++;
            cout<< current ->data;
            current = current->next;
      }
      return count;
}

int main() {
      struct node* myList = BuildOneTwoThree();
      int len = Length(myList); // results in len == 3
}
```

main

assume this function creates 3 nodes on the heap

Stack

10
myList

len

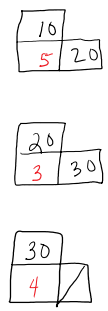
length

10
head

10
Current

0
Count

heap



initial values when you enter the function

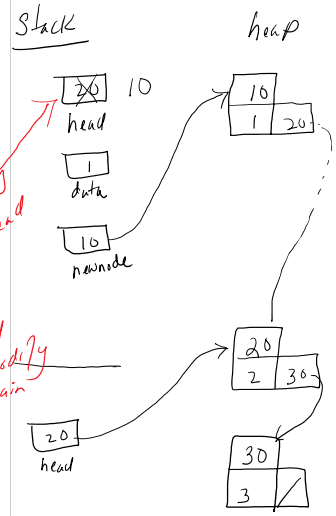
Example 5 (Adding to beginning: Error with linked list - local variable)
The key is that the line `head = newNode;` changes the `head` local in wrongAdd() but not the `head` back in main().

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

struct node* build() {
      struct node* head;

      head = malloc(sizeof(struct node)); // allocate on the heap
      head->data = 2;       // setup first node
      head->next = malloc(sizeof(struct node)); // allocate on the heap
      head->next->data = 3;     // setup second node
      head->next->next = NULL;

      return (head);
}

//The change to head is not passed back
void wrongAdd (struct node* head, int data) {
      struct node* newNode = (node *)malloc(sizeof(struct node));
      newNode->data = data;
      newNode->next = head;
      head = newNode; // NO this line does not work!
}

int main() {
      struct node* head = build();
      wrongAdd(head, 1); // try to add 1 to the front -- doesn't work
      cout<<head->data;
}
```

Wrong add

Changes only this head

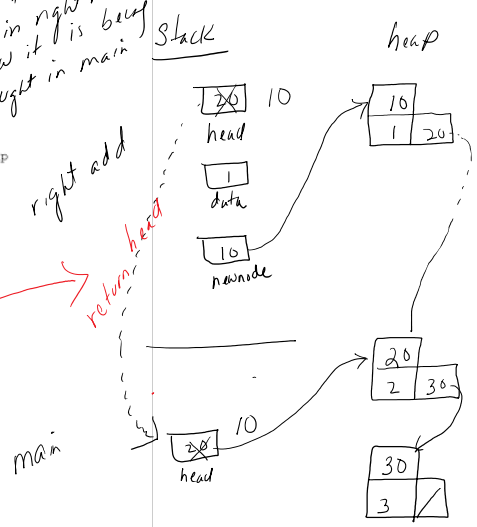
Changes the local head. Does not modify the head in main



Example 6 (Corrected version of pervious example)

```
//same as before
struct node {
  int data;
  node *next;
};

//same as before
struct node* build() {
      struct node* head;

      head = malloc(sizeof(struct node)); // allocate on the heap
      head->data = 2;      // setup first node
      head->next = malloc(sizeof(struct node)); // allocate on the heap
      head->next->data = 3;    // setup second node
      head->next->next = NULL;

      return (head);
}

//modified: Returns head
struct node* rightAdd (struct node* head, int data) {
      struct node* newNode = (node *)malloc(sizeof(struct node));
      newNode->data = data;
      newNode->next = head;
      head = newNode;
      return head
}

//modified: catches head after rightAdd call
int main() {
      struct node* head = build();
      head= rightAdd (head, 1);
      cout<<head->data;
}
```

This is exactly like the previous example except for what is being returned in rightAdd and how it is being caught in main



Example 7 (Correcting the previous example again using **)

- Design the function to take a pointer to the head pointer. This is the standard technique in C — pass a pointer to the "value of interest" that needs to be changed. To change a `struct node*`, pass a `struct node**`.
- Use '&' in the caller to compute and pass a pointer to the value of interest.
- Use '*' on the parameter in the callee function to access and change the value of interest.

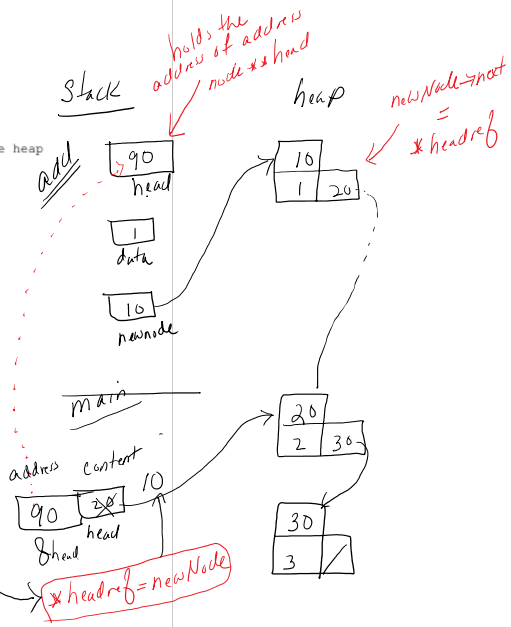
nothing is passed back because of node ** headref in add function

```
/*
Takes a list and a data value. Creates a new link with the given data and
pushes it onto the front of the list. The list is not passed in by its head
pointer. Instead the list is passed in as a "reference" pointer to the head
pointer -- this allows us to modify the caller's memory.
*/

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

struct node* build() {
      struct node* head;

      head =(node *) malloc(sizeof(struct node)); // allocate on the heap
      head->data = 2;       // setup first node
      head->next = (node *)malloc(sizeof(struct node));
      head->next->data = 3;     // setup second node
      head->next->next = NULL;

      return (head);
}
void add(struct node** headRef, int data) {
      struct node* newNode = (node *)malloc(sizeof(struct node));
      newNode->data = data;
      newNode->next = *headRef;
      *headRef = newNode;
}

void main() {
      struct node* head = build();
      add (&head, 1); // note the &
      add (&head, 13);
      // head is now the list {13, 1, 2, 3}
}
```

no need to return anything

dereferencing variable

two nodes are being added. Notice the & in the call



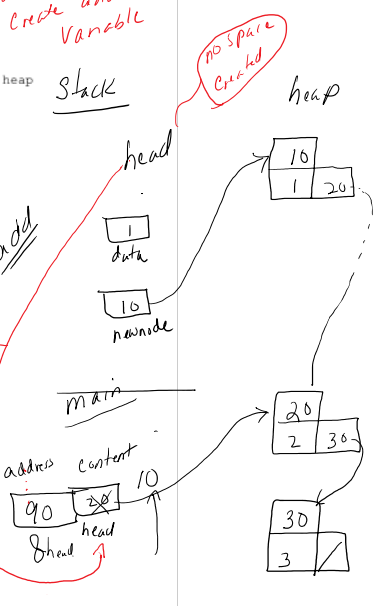
Example 8 (Correcting the previous example using &)

```
struct node* build() {
      struct node* head;

      head = malloc(sizeof(struct node)); // allocate on the heap
      head->data = 2;       // setup first node
      head->next = malloc(sizeof(struct node)); // allocate on the heap
      head->next->data = 3;    // setup second node
      head->next->next = NULL;

      return (head);
}

//The change to head is not passed back
void add (struct node* &head, int data) {
      struct node* newNode = (node *)malloc(sizeof(struct node));
      newNode->data = data;
      newNode->next = head;
      head = newNode;
}

int main() {
      struct node* head = build();
      add(head, 1);
      cout<<head->data;
}
```

ampersand in the function header does not create another variable

no space created

no storage space created for head in function add. Identifier in add function behaves in exactly the same way as head in main



Example 9 (Adding to the end)

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

struct node* build() {
       struct node* head;

       head = (node *)malloc(sizeof(struct node)); // allocate on the heap
       head->data = 2;
       head->next = (node *)malloc(sizeof(struct node));
       head->next->data = 3;    // setup second node
       head->next->next = NULL;

       return (head);
}

void addToEnd (struct node * current, int data){
     struct node* newNode;

     while ( current->next != NULL)
           current = current->next;

    newNode = new node;      // Sets it to actually point to something
    newNode->data = data;
    newNode->next = NULL;
    current->next=newNode;
}


int main (){
    struct  node * head=build();
    if (head) //Make sure head is valid
            addToEnd(head, 5);

    while ( head != NULL ) {
          cout<< head->data;
          head=head->next;
    }
}
```

go to the last node in the list

Create a node a Connect to the end of list

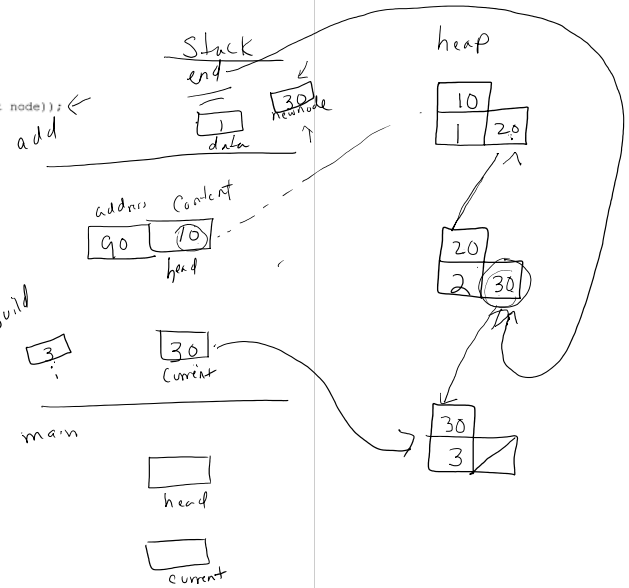
Create 2 nodes on the heap

if head is defined add node to the end

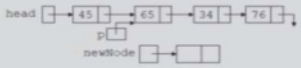
display the data elements of all nodes in the list

Example 10 (Adding to the end, another example)

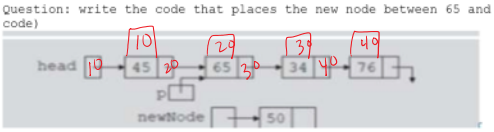
```
struct node {
  int data;
  node *next;
};
typedef struct node node;

node* add(node* end, int data) {
     node* newNode = (node *)malloc(sizeof(struct node));
     newNode->data = data;
     newNode->next = NULL;
     end = newNode;
     return (end);
}

node* build () {
   node* head, *current;

   head=current=NULL;

   for (int i=1; i<4; i++) {
      if (!head)
          current=add(head, i);
      else
          current=add(current->next, i);
   }

   return(head);
}

int main(){
   node* head, *current;

   current=head=build ();
   while (current!= NULL ) {
          cout<< current ->data;
          current = current ->next;
   }
```



Adding to the middle (Visual)

Statement	Effect
`newNode = new nodeType;`	
`newNode->info = 50;`	
`newNode->link = p->link;`	
`p->link = newNode;`	

Question: write the code that places the new node between 65 and 34 (Hard code)



Question: Write the code that points you to the last node (Hard code)

Question: Write the code that points you to the last node (Generic code)