CS32: Introduction to Computer Science II

Discussion Week 3

Yichao (Joey)

April 17, 2019

Credit: Carey Nachenberg, Junheng Hao

Announcements

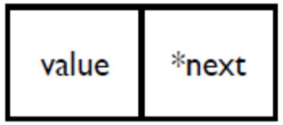
- Project 2 is due on 11:00 PM Wednesday, April 22

Outline Today

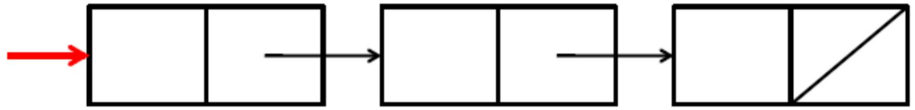
- Linked List
- Double Linked List
- Circular vs Linked List with Loop

Linked List: Review
Basis

- Minimum Requirement
  - Key component as unit: Node (with `value` and `pointer` to next node)
  - Head pointer → points to the first term
- Regular operations
  - Insertion
  - Search
  - Removal
- Pros and cons
  - Efficient insertion, flexible memory allocation, simple implementation
  - High complexity of search



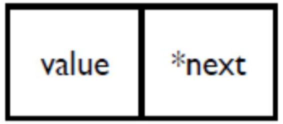
```
typedef int ItemType;
Struct Node
{
      ItemType value;
      Node *next;
};
```



Linked List: Review

Basis

- Drawing pictures and carefully tracing through your code, updating the picture with each statement, can help you find bugs in your code.
- Check any list operations for these: (Does it work correctly)
  - in a typical, middle-of-the-list case?
  - at the beginning of the list?
  - at the end of the list?
  - for the empty list?
  - for a one-element list?
- Another validation technique is for every expression of the form p->something, prove that you can be sure p has a well-defined, non-null value at that point.

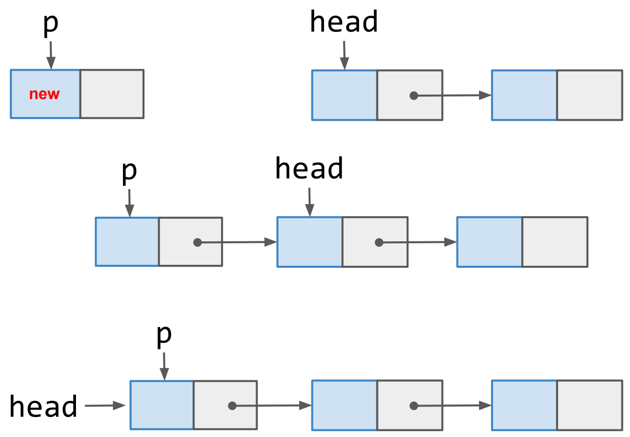


```
typedef int ItemType;
Struct Node
{
      ItemType value;
      Node *next;
};
```

Linked List

Insertion: Add a new node to a list

- Example: Insert as head in a list
- Steps
  a) Create a new node and call the pointer p
  b) Make its `next` pointer point to the first item
  c) Make the head pointer to the new node



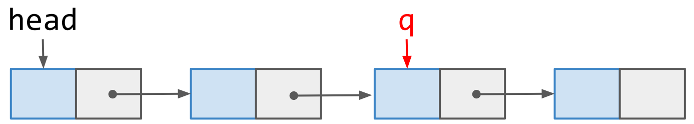
```
//Skeleton: Linked list insertion
//=======================================

//insert as head
p->next = head;
Head = p;

//insert after end: End node: q
q->next = p;
p->next = nullptr;

//insert in the middle: node q
p->next = q->next;
q->next = p;
```

Linked List

Search

UCLA Samueli Computer Science

- Steps
  a) Find matched node and return
  b) If no match, return NULL

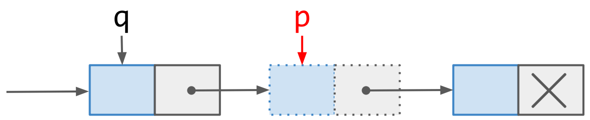


```
// Skeleton Code: Linked list search
// =======================================


Node* Search(int key, Node* head){
  Node *q = head;
  while(q != NULL)
  {
    if(q -> value != key) q = q -> next;
    else return q;
  }
  return NULL;
}
```

Linked List

Removal

- Remember to set the previous node q's `next` pointer to point the next node of p
  `q->next = p->next;`
  `delete p`
- What if `p == head`? What if `p` prints to the last node in the linked list?



```
// Skeleton Code: Linked list removal
// ========================================

void remove(int valToRemove, Node* head) {
      Node *p = head, *q = NULL;
      while (p != NULL) {
            if (p->value == valToRemove)
                  break;
            q = p;
            p = p->next;}
      if (p == NULL) return;
      if (p == head) //special case
            head = p->next;
      else
            q->next = p->next;
      delete p;
}
```

Linked List
Conclusion

- Pros:
  - Efficient insertion (add new data items)
  - Flexible memory allocation
- Cons:
  - Slow search (search is more important than insertion and removal in real situations)
    - e.g. retrieve the fifth value of the list.
    - e.g. a list of values is sorted, find 10 in the linked list
- Many variations
  - Doubly linked lists
  - Sorted linked lists
  - Circularly linked lists

Problem: Reverse Linked List

Leetcode questions #206

Question: How to reverse a (single) linked list?

Example:

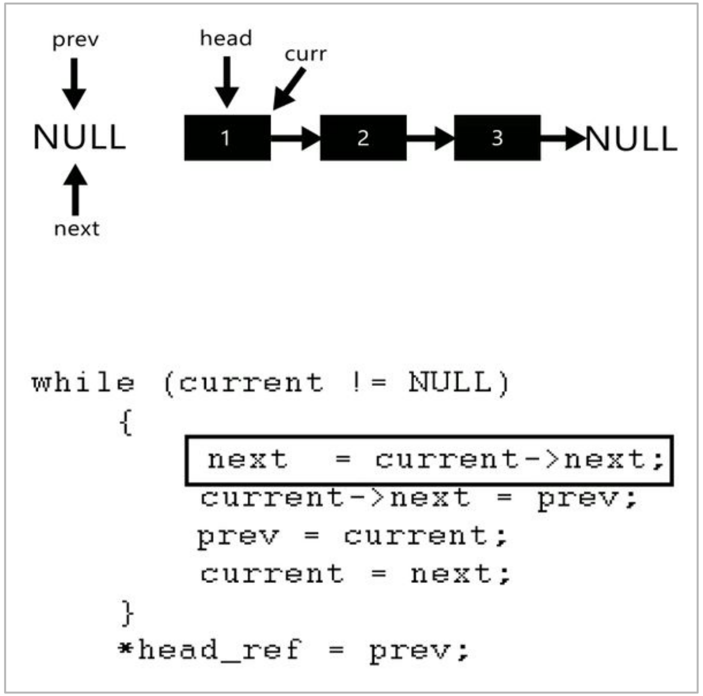
```
Input: 1->2->3->4->5->NULL
Output: 5->4->3->2->1->NULL
```

```
// One possible solution
Node* reverseList(Node* head)
{
     Node *prev=NULL,*current=head,*next;
     while(current) {
          next = current->next;
          current->next = prev;
          prev = current;
          current = next;
     }
     return prev;
}
```

Problem: Reverse Linked List

Leetcode questions #206

Let's see what happens in these lines of codes! [Link]



```
while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
```

Problem: Reverse Linked List

Leetcode questions #234

Given a singly linked list, determine if it is a palindrome.

Example 1:

```
Input: 1->2
Output: false
```

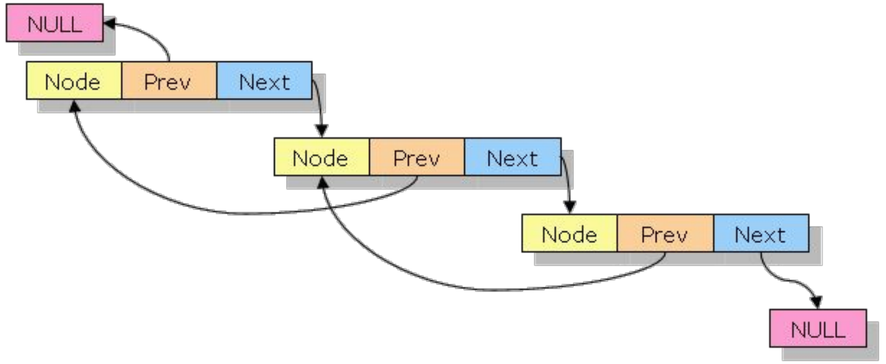
Example 2:

```
Input: 1->2->2->1
Output: true
```

```
// One possible solution (Pseudo Code)

Reverse the linked list A -> A_rev
Traverse each node of both linked list:
     If (node_1.value in A doesn’t equal to
node_2.value in A_rev):
          Return False
     Else:
          Continue
Return True
```

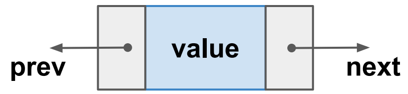
Double Linked List

Data structures and properties



- A linked list where each node has two pointers:
  - Next – pointing to the next node
  - Prev – pointing to the previous node
- Features
  - head, tail pointers
  - `head->prev = NULL; tail->next = NLL;`
  - `head == tail == NULL` when doubly linked list is empty

```
typedef int ItemType;
Struct Node
{
        ItemType value;
        Node *next;
        Node *prev;
};
```



Double Linked List

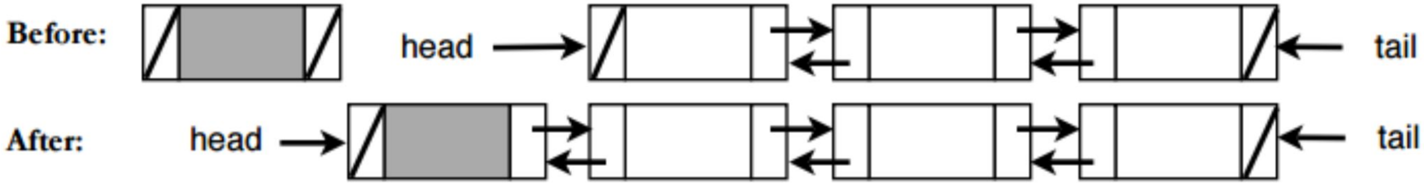
Insertion: How many cases to consider?

- Four cases:
  - Insert before the head
  - Insert after the tail
  - Insert somewhere in the middle
  - When list is empty

Double Linked List

Insertion: Before head / After tail

- Steps for insertion before head:
  - Set the `prev` of head to the new node p
  - Set the `next` of p to `head`
  - p becomes the new `head`
  - `head->prev = NULL;`
- Steps for insertion after tail:
  - Similar to insertion before head (try it yourself!)



Double Linked List

Insertion: In the middle of the list

- Steps for insertion in the middle (new node p) (insert after node q):
  - Fix the next node of q first: `Node *r = q->next;`
  - Point both next of q and prev of r to p: `q->next = r->prev = p;`
  - Point both sides of p to q and r respectively: `p->prev = q; p->next = r;`

- You can do that without the help of pointer `r`

```
p->prev = q;
p->next = q->next;
q->next = q->next->prev = p;
```

Double Linked List

Insertion to empty list / Search

- Insertion to an empty list

```
head = tail = p;
p->next = p->prev = NULL;
```

- Search in doubly linked list
  - Similar to standard linked list
  - Can be done either from head or tail

Double Linked List

Removal

- Removal is more complex!
- Consider the following cases:
    - Check if the node p is the head `(p == head)`. Let this boolean be A.
    - Check if the node p is the tail `(p == tail)`. Let this boolean be B.
- Different cases:
    - Case 1 (A, but not B): P is the head of the list and there is more than one node.
    - Case 2 (B, but not A): P is the tail of the list, and there is more than one node.
    - Case 3 (A and B): P is the only node.
    - Case 4 (not A and not B): P is in the middle of the list.

Double Linked List

Removal

UCLA Samueli Computer Science

```
void removeNodeInDLL(Node *p, Node& *head, Node& *tail)
{
      if (p == head && p == tail) //case 3
            head = tail = NULL;
      else if (p == head) {
            //case 1
            head = head -> next;
            head -> prev = NULL; }
      else if (p == tail) {
            //case 2
            tail = tail -> prev;
            tail -> next = NULL; }
      else {
            //case 4
            p -> prev -> next = p -> next;
            p -> next -> prev = p -> prev; }
      delete p;
}
```

Double Linked List

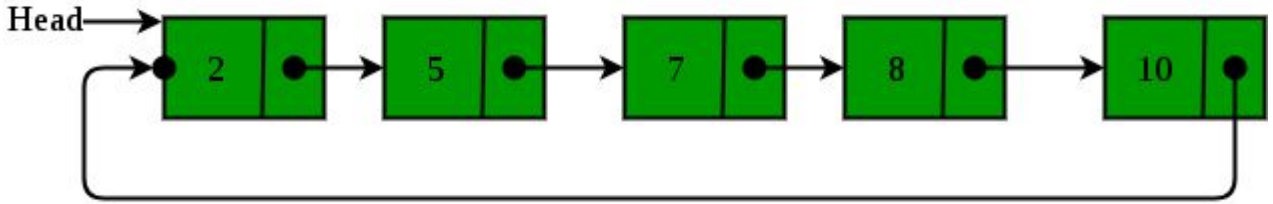
Copy a doubly linked list (and more)

- Steps
  - Create head and tail for the new list
  - Iterate through the old list. For each node, copy its value to a new node.
  - Insert the new node to the tail of the new list.
  - Repeat until we have iterated the entire old list.
  - Set `NULL` before head and next of tail.

- Tips for linked list problems
  - To draw diagrams of nodes and pointers will be extremely helpful.
  - When copying a linked list, only copy stored values to new nodes. Do not copy pointers.
  - You need to check **edge cases**!

Circular Linked List

Motivation and properties

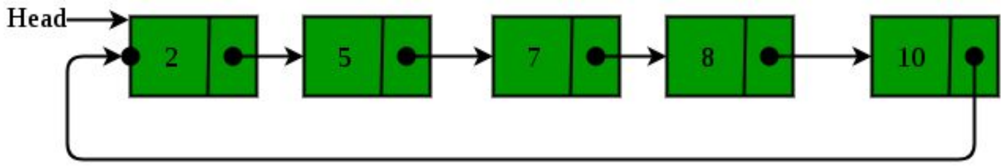


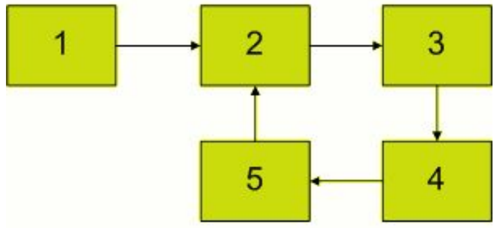
- Linked list where all nodes are connected to form a circle.
    - There is no NULL at the end.
    - Can be a singly circular linked list or doubly circular linked list.
- Pros:
    - Any pointer can be head (starting point).
    - Implementation for queue.
    - Fit to repeatedly go around the list.

We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

Circular vs Linked List with Loop

- Two different tasks:
  - Tell whether the linked list is circular
  - Tell whether there is a loop in the linked list (this is much harder!)



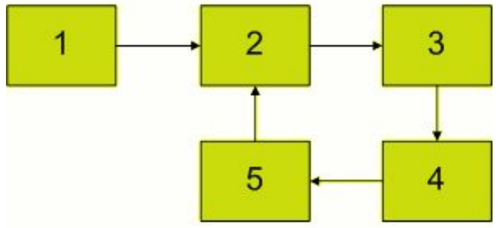


Circular vs Linked List with Loop

UCLA Samueli Computer Science

- Two different tasks:
  - Tell whether the linked list is circular
  - Tell whether there is a loop in the linked list (this is much harder!)

```
 bool hasCycle(Node* head) {
    Node *slow_ptr = head, *fast_ptr = head;
    while (fast_ptr && fast_ptr->next) {
      slow_ptr = slow_ptr->next;
      fast_ptr = fast_ptr->next->next;
      // compare slow_ptr and fast_ptr at least
after 1 update
      if (slow_ptr == fast_ptr) return true;
    }
    return false;
  }
```



Suggestions on Linked List

- Drawing pictures!!!
- Check any list operations for these:
  - Middle
  - Beginning
  - End
  - Empty
  - One-element
- `p->something`

Group Exercises: Worksheet

- Exercise problems from **Worksheet 2** (see “LA worksheet” tab in CS32 website). Answers will be posted after all discussions.
- Questions for today:
    - #2: Linked List Copy Constructor
    - #3: Find Nth from Last
    - #4: Rotate Left by N

Group Exercises: Worksheet Prob. #2

Implement the **copy constructor**: create a **new linked list** with the **same number of nodes** and **same values.**

```
class LL {
      public:
            LL() { head = nullptr; }
            LL(const &other){
                  // Implement copy constructor
            }
      private:
            struct Node {
            public:
                  int val;
                  Node* next;
            };
            Node* head;
};
```

Group Exercises: Worksheet Prob. #2

```
LL(const LL& other) {
      if (other.head == nullptr)
            head = nullptr;
      else {
          head = new Node;
          head->val = other.head->val;
          head->next = nullptr;
          Node* thisCurrent = head;
          Node* otherCurrent = other.head;

          // dynamically allocate new list based on other
          while (otherCurrent->next != nullptr) {
            thisCurrent->next = new Node;
            thisCurrent->next->val = otherCurrent->next->val;
            thisCurrent->next->next = nullptr;

            thisCurrent = thisCurrent->next;
            otherCurrent = otherCurrent->next;
            }
      }
}
```

Group Exercises: Worksheet Prob. #3

Using the same class LL from the previous problem, write a function *findNthFromLast* that returns the **value** of the Node that is **nth from the last Node** in the linked list.

```
int LL::findNthFromLast(int n);
```

Approaches:

- Using size of linked list
- Can we do it without finding size?

Group Exercises: Worksheet Prob. #3

```
int LL::findNthFromLast(int n) {
  Node* p = head;
  for (int i = 0; i < n; i++) {
    if (p == nullptr) {
      return -1;
    }
    p = p->next;
  }
  if (p == nullptr) {
    return -1;
  }

  Node* nthFromP = head;
  while (p->next != nullptr) {
    p = p->next;
    nthFromP = nthFromP->next;
  }
  return nthFromP->val;
}
```

Overview:

- Move initial pointer **n** nodes forward
- Initialize 2nd pointer at head
- When the initial pointer reaches the end, the 2nd pointer will be **n** from the end

Group Exercises: Worksheet Prob. #4

Write a function *rotateLeft* function such that it **rotates the linked list to the left, *n* times**. Rotating a list left consists of shifting elements left, such that elements at the front of the list loop around to the back of the list. The new start of the list should be stored within *head*.

```
struct Node {
      int val;
      Node* next;
};

class LinkedList {
public:
      void rotateLeft(int n);
private:
      Node* head;
};
```

Group Exercises: Worksheet Prob. #4

```
void LinkedList::rotateLeft(int n) {
  if(head == nullptr)
    return;
  int size = 1;
  Node* oldTail = head;
  while (oldTail->next != nullptr) {
    size++;
    oldTail = oldTail->next;
  }
  if (n % size > 0) {
    int headPos = n % size;
    Node* newTail = head;
    for (int x = 0; x < headPos - 1; x++) {
      newTail = newTail->next;
    }
    Node* newHead = newTail->next;
    newTail->next = nullptr;
    oldTail->next = head;
    head = newHead;
  }
}
```

Overview:

- Get **size** and **tail** of linked list
- Get the **new tail** after rotating
- Get the **new head**
- Mark end of the new list with nullptr
- Link old tail to old head
- Set new head