CS32 Discussion
Week 3

Muhao Chen
muhaochen@ucla.edu
http://yellowstone.cs.ucla.edu/~muhao/
Outline

• Doubly Linked List
• Sorted Linked List
• Reverse a Linked List
Doubly Linked List

• A linked list where each node has two pointers:
  • Next – pointing to the next node
  • Prev – pointing to the previous node

• struct Node {
    int value;
    Node *Next;
    Node *Prev;
• };
Doubly Linked List

• That’s how it looks like:

• Features to capture a DLL:
  • Two pointers: head, tail
  • head -> prev = NULL
  • tail -> next = NULL
  • head == tail == NULL when list is empty
Insertion

Four different conditions to insert a new node $P$
1. Insert before the head;
2. Insert after the tail;
3. Insert somewhere in the middle
4. When list is empty;
Insertion (Before head)

1) Set the `prev` of `head` to the new node `p`
   - `head -> prev = p;`
2) Set the `next` of `p` to `head`
   - `p -> next = head;`
3) `p` becomes the new `head`
   - `head = p;`
4) `head -> prev = NULL;`
Insertion (after tail)

• Quite the same as insertion before head:

```c
    tail -> next = p;
p -> prev = tail;
tail = p;
tail = p;
p -> next = NULL;
```
Insertion in the middle (after node \(q\))

1) Fix the next node of \(q\) first:
   - Node *\(r = q -> next;\)
2) Point both next of \(q\) and prev of \(r\) to \(p\)
   - q -> next = r -> prev = p;
3) Point both sides of \(p\) to \(q\) and \(r\) respectively:
   - p -> prev = q;
   - p -> next = r;
Insertion in the middle *(after node q)*

• Or do it without r:
  • $p \rightarrow \text{prev} = q$;
  • $p \rightarrow \text{next} = q \rightarrow \text{next}$;
  • $q \rightarrow \text{next} = q \rightarrow \text{next} \rightarrow \text{prev} = p$;
Insertion (to an empty list)

• How do we represent an empty list?
  • head == NULL (Or tail == NULL; Or head == tail == NULL)

• 1) Insertion, just set $p$ as head (as well as tail):
  • head = tail = $p$;

• 2) Don’t forget to set NULL on both sides:
  • $p$->next = $p$->prev = NULL;
Search

• Just like the singly linked list.

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

Node* Search(int key, Node* tail){
    Node *q = tail;
    while(q != NULL) {
        if(q -> value == key) return q;
        else q = q -> prev; //iterate to the previous node
    }
    return NULL;
}
Removal

• More complex than singly linked list.
  • Check if the node $p$ is the head ($p == \text{head}$). Let this boolean be A.
  • Check if the node is the tail ($p == \text{tail}$). Let this boolean be B.
Removal

• Four 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.
Removal Case 1 \((P\text{ is head})\)

- **1) Update** \(head\)
  - head = head -> next;

- **2) delete** \(p\)
  - delete p;

- **3) Set the** \(prev\) **of** \(head\) **to** NULL
  - head -> prev = NULL;
Removal Case 2 ($P$ is tail)

1) Update $tail$
   - $tail = tail -> prev$;

2) delete $p$

3) Set the next of $tail$ to NULL
   - $tail -> next = NULL$;
Removal Case 3 \((P \text{ is the only node})\)

1. Empty the linked list:
   - head = tail = NULL;
2. delete p:
Removal Case 4 \((P \text{ is in the middle})\)

1) Fix the \textit{prev} and \textit{next} of \(p\):
   - Node *q = p -> prev;
   - Node *r = p -> next;

2) Concatenate \(q\) and \(r\):
   - q -> next = r;
   - r -> prev = q;

3) Delete \(p\)
Removal Case 4 (Equivalent implementation)

• If we do not fix with $q$ and $r$:
  • $p \rightarrow \text{prev} \rightarrow \text{next} = p \rightarrow \text{next}$;
  • $p \rightarrow \text{next} \rightarrow \text{prev} = p \rightarrow \text{prev}$;
  • delete $p$;
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;
}
Copying a doubly linked list

• 1) Create head and tail for the new list
• 2) Iterate through the old list. For each node, copy its value to a new node.
• 3) Insert the new node to the tail of the new list.
• 4) Repeat 3 until we have iterated the entire old list. Set NULL before head and next of tail.
void copyDDL(Node *head_o, Node *tail_o, Node& *head_n, Node& *tail_n) {
    if (tail_o == NULL) { //the original list is empty
        head_n = tail_n = NULL; return;
    }
    Node *q = head_o; //iterator
    Node *p = new Node();
    p -> value = q -> value;
    head_n = tail_n = p;
    q = q -> next;
    while (q) {
        p = new Node();
        p -> value = q -> value;
        tail_n -> next = p;
        p -> prev = tail_n;
        tail_n = tail_n -> next;
        q = q -> next;
    }
    head_n -> prev = tail_n -> next = NULL
}
Cautions about coding with a linked list

- To draw diagrams of nodes will be extremely helpful.
- When copying a linked list, only copy stored values to new nodes. Do not copy pointers.
Sorted linked list
Do we need to search the entire linked?

• Consider an ascending sorted (doubly linked) list:

• Do we need to search through all the nodes when we’re searching for:
  • 8
  • 50

• A way to optimize the search: sorted list and early stop
How to implement an ascending sorted (doubly linked) list?

• Change the insertion. Find the node $q$ whose value is the greatest lower bound to the new node $p$.

• 1) Check if $head$’s value is larger than $p$’s. If so, insert $p$ as head.
   
   ```
   if (p -> value < head -> value) {
       p -> next = head;
       head -> prev = p;
       p -> prev = NULL;
       head = p;
   }
   ```
How to implement an ascending sorted (doubly linked) list?

2) if not, iterate through the list until we find the node \( q \) whose value is the greatest lower bound to \( p \).

- Note: \( q \rightarrow \text{next} \) can either have a value larger than \( q \), or is \( \text{NULL} \).

```c
Node *q = head -> next;
while (q && q -> value < p -> value) {
    if (q-> next == NULL || q -> next -> value > p -> value) break;
    q = q -> next;
}
```
How to implement an ascending sorted (doubly linked) list?

• 3) Insert $p$ after $q$:

```c
if (q -> next != NULL)
    q -> next -> prev = p;
p -> next = q -> next; /* if q is the last node then its next is already NULL */
p -> prev = q;
q -> next = p;
```
void insert(Node *p, Node *head) {
    if (p -> value < head -> value) {
        p -> next = head;
        head -> prev = p;
        p -> prev = NULL;
        head = p;
        return;
    }
    Node *q = head -> next;
    while (q -> value < p -> value) {
        if (q-> next == NULL || q -> next -> value > p -> value) break;
        q = q -> next;
    }
    if (q -> next != NULL)
        q -> next -> prev = p;
    p -> next = q -> next;
    p -> prev = q;
    q -> next = p;
}
Early stop in searching a sorted linked list

We stop the iteration once we see a node which stores a value that is larger than key.

Node* Search(int key, Node* head){
    Node *q = head;
    while(q != NULL && q -> value <= key) {
        if(q -> value == key) return q;
        else q = q -> next; //iterate to the next node
    }
    return NULL;
}
What about removal and update?
Why *early stop* technique saves cost

• Spare cost from search to insertion and update
• However, search is called massively, but insertion and updates are not.
• $O(n/2)$ cost is saved for each search (assuming the data complies with a uniform distribution)
Reverse Linked List (Leetcode #206, easy)

Given a singly linked list, reverse every node of it (i.e. each next points to the previous node).

Node* reverseList(Node* head)
Solution

Node* reverseList(struct ListNode* head) {
    Node *prev=NULL,*cur=head,*next;
    while (cur) {
        next = cur->next;
        cur->next = prev;
        prev = cur;
        cur = next;
    }
    return prev;
}
Bugs in your software are actually special features :)