Linked Lists

David Yue

Linked Lists: Introduction

- A data structure that involves chaining together elements through a pointer from the previous element.
- eg: A linked list data struct to store a list of integers.

```
struct linkedlist_node {
  int item;
  struct linkedlist_node *next;
}
```

- Last item in list has `next = NULL`

Linked Lists: Creating a node

- Typically linked list nodes are all allocated on the heap
  - Does not make sense to have linked list nodes on the stack, just use an array instead.
- Linked lists are passed around as a pointer to their first element

Linked Lists: Creating a node

- Typically linked list nodes are all allocated on the heap
  - Does not make sense to have linked list nodes on the stack, just use an array instead.
- Linked lists are passed around as a pointer to their first element

```
struct linkedlist_node* create_node(int value) {

  struct linkedlist_node *new_node = \
      malloc(sizeof (struct linkedlist_node));

  if (!new_node)
      return NULL;

  new_node->next = NULL;
  new_node->item = 0;

  return new_node;
}
```

Linked Lists: Insertion

- How do we insert a node? Simply modify an existing node in the list to point to the new node.

Linked Lists: Insertion

- How do we insert a node? Simply modify an existing node in the list to point to the new node.
- Here is an example of inserting a new node into the end of the linked list.

```
void insert_at_end(struct linkedlist_node* list, \
                   struct linkedlist_node* node) {
    // assumes list != NULL
    while (list->next)
        list = list->next;

    list->next = node;

}
```

Linked Lists: Insertion

- What if we want to insert at the beginning? Or insert into an *ordered list*?
- Naive Solution: Return a pointer to the new head of the list.

Linked Lists: Insertion

- What if we want to insert at the beginning? Or insert into an *ordered list*?
- Naive Solution: Return a pointer to the new head of the list.

```
struct linkedlist_node* \
insert_at_begin(struct linkedlist_node* list, \
                struct linkedlist_node* node) {

    node->next = list;

    return node;
}
```

Linked Lists: Insertion

```
struct linkedlist_node* \
insert_ordered(struct linkedlist_node* list, \
               struct linkedlist_node* node) {

    struct linkedlist_node *prev = NULL;
    struct linkedlist_node *curr = list;
    while (curr->item < node->item) {
        prev = curr;
        curr = curr->next;
    }

    if (prev)
        prev->next = node;

    node->next = curr;

    return prev ? list : node;
}
```

Linked Lists: Insertion

- Good, but I don't like having different return types depending on how we wish to insert into the list.
- Better Solution: Use a double pointer to the list instead. No return value needed.

Linked Lists: Insertion

```
void insert_ordered(struct linkedlist_node** list, \
                    struct linkedlist_node* node) {

    struct linkedlist_node *prev = NULL;
    struct linkedlist_node *curr = *list;
    while (curr->item < node->item) {
        prev = curr;
        curr = curr->next;
    }

    node->next = curr;

    if (prev)
        prev->next = node;
    else
        *list = node;

}
```

Linked Lists: Insertion

- Good, but I don't want to keep track of the previous element!
- Even better solution: Utilize the double pointer to keep track of previous instead

Linked Lists: Insertion

- Good, but I don't want to keep track of the previous element!
- Even better solution: Utilize the double pointer to keep track of previous instead

```
void insert_ordered(struct linkedlist_node** list, \
                    struct linkedlist_node* node) {

    while ((*list)->item < node->item)
        list = &((*list)->next);

    node->next = *list;
    *list = node;
}
```

Linked Lists: Deletion

- How do we delete from a linked list?
- Simply unlink the node from the list

Linked Lists: Deletion

```
struct linkedlist_node* \
delete_item(struct linkedlist_node* list, int value) {

    struct linkedlist_node *prev = NULL;

    while (list && list->item != value) {
        prev = list;
        list = list->next;
    }

    struct linkedlist_node *new_head = list;

    if (prev)
        prev->next = list->next;
    else
        new_head = list->next;

    free(list);

    return new_head;
}
```

Linked Lists: Deletion

- Deletion also requires us to potentially modify the head element
  - Use a double pointer
- Deletion requires us to keep track of previous element
  - Utilize double pointer trick from `insert_ordered`
- Left as an exercise