lec10-linkedlist

April 16, 2018

1 Lecture 10: Linked list

Zewei Chu 4/16/2018

1.0.1 Announcement

Exam 1 is this Friday, 4/20, 2:30 in this classroom

1.1 Linked List

A linked list is a linear data structure where each element is a separate object. Each element (we will call it a node) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the head of the list.

In [36]:
```
#include<stdio.h>
typedef struct _node node;
struct _node{
    int value;
    node *next;
};

void print_intlist(node* p){
    while (p != NULL){
        printf("%d ", p->value);
        p = p->next;
    }
}
int main(){
    node *p1, *p2, *p3, *p4;
    node l1, l2, l3, l4;
    p1 = &l1;
    p2 = &l2;
    p3 = &l3;
    p4 = &l4;

    l1.value = 1;
    l2.value = 2;
    l3.value = 3;
```

```
    l4.value = 4;

    p1->next = p2;
    p2->next = p3;
    p3->next = p4;
    p4->next = NULL;

    print_intlist(p1);
    return 0;
}
```

```
1 2 3 4
```

- Usually we use malloc to dynamically create linked list.
- free the linked list at the end to prevent memory leak.

1.1.1 Linked List Inspection

In [37]:
```
#include<stdio.h>
#include<stdlib.h>
typedef struct _node node;
struct _node{
    int value;
    node *next;
};

// print information of a node
void print_node(node* p){
    printf("node: %d\n", p->value);
}

// print the whole int list
void print_intlist(node* p){
    while (p != NULL){
        printf("%d ", p->value);
        p = p->next;
    }
    printf("\n");
}

// free the whole int list
void free_intlist(node* p){
    node* tmp;
    while (p != NULL){
        tmp = p;
        p = p->next;
    }
}
```

```c
// raise an error
void raise_error(char* msg, int code){
    fprintf(stderr, "%s\n", msg);
    exit(code);
}

// check if the linked list is empty
int is_empty(node* p){
    return p==NULL;
}

// return the first element of the linked list
node* first(node* p){
    if (is_empty(p)){
        raise_error("The linked list is empty", -1);
    }
    return p;
}

// return all but the first element in the linked list
node* rest(node* p){
    if (is_empty(p) || is_empty(p->next)){
        raise_error("The linked list has less than 2 elements", -2);
    }
    return p->next;
}

// return sum of the int linked list
int sum(node* p){
    int res = 0;
    for (; p!=NULL; p=p->next){
        res += p->value;
    }
    return res;
}

// return sum of the int linked list with recursion
int sum_rec(node* p){
    if (p == NULL)
        return 0;
    return p->value + sum_rec(p->next);
}

// add v to the beginning of the linked list
// return the head node pointer
node* push_front(int v, node *list){
    node* p = (node*)malloc(sizeof(node));
```

```
    p->value = v;
    p->next = list;
    return p;
}

// node_create  create a list containing length nodes.
// Return pointer to first node.
// Write iteratively using push_front
node* node_create_push_front(int length){
    node* head = NULL;
    for (int i = 0; i < length; ++i)
        head = push_front(0, head);
    return head;
}

// create a linked list containing length nodes
// without using push_front
node *node_create(int length){
    node* head = (node*)malloc(sizeof(node));
    head->value = 0;
    node* curr = head;
    for (int i = 1; i < length; ++i){
        curr->next = (node*)malloc(sizeof(node));
        curr = curr->next;
        curr->value = 0;
    }
    curr->next = NULL;

    return head;
}

// insert a node at the end of a linked list
node* push_back(int v, node* p){
    if (is_empty(p)){
        raise_error("The linked list is empty", -1);
    }
    node* tmp = p;
    while (tmp->next != NULL){
        tmp = tmp->next;
    }
    tmp->next = push_front(v, tmp->next);
    return p;
}

node* insert_at(int v, node* p, int pos){
    node* newnode = (node*)malloc(sizeof(node));
    newnode->value = v;
    if (is_empty(p)){
```

```
        return newnode;
    }
    if (pos == 0){
        newnode->next = p;
        return newnode;
    }
    node* tmp = p;
    int i = 1;

    for (; tmp->next!=NULL && i < pos; ++i, tmp=tmp->next)
        ;
    newnode->next = tmp->next;
    tmp->next = newnode;
    return p;
}

int main(){
    node *head, *current;
    head = current = (node*)malloc(sizeof(node));
    current->value = 0;
    current->next = (node*)malloc(sizeof(node));
    current = current->next;
    current->value = 1;
    current->next = (node*)malloc(sizeof(node));
    current = current->next;
    current->value = 2;
    current->next = NULL;

    print_intlist(head);
    printf("Is the linked list empty? %d\n", is_empty(head));

    node* f = first(head);
    print_node(f);

    node* r = rest(head);
    print_intlist(r);

    int s = sum(head);
    printf("sum: %d\n", s);

    s = sum_rec(head);
    printf("sum with recursion: %d\n", s);

    head = push_front(3, head);
    print_intlist(head);

    node* head2 = node_create(10);
    print_intlist(head2);
```

```
    head = push_back(4, head);
    print_intlist(head);

    head = insert_at(5, head, 0);
    print_intlist(head);

    head = insert_at(6, head, 1);
    print_intlist(head);

    head = insert_at(7, head, 2);
    print_intlist(head);

    head = insert_at(8, head, 8);
    print_intlist(head);

    head = insert_at(9, head, 20);
    print_intlist(head);

    free_intlist(head);
    free_intlist(head2);

    return 0;
}
```

Advantages of linked list

- Items can be added or removed from the middle of the list
- There is no need to define an initial size

Disadvantages

- There is no "random" access - it is impossible to reach the nth item in the array without first iterating over all items up until that item. This means we have to start from the beginning of the list and count how many times we advance in the list until we get to the desired item.
- Dynamic memory allocation and pointers are required, which complicates the code and increases the risk of memory leaks and segment faults.
- Linked lists have a much larger overhead over arrays, since linked list items are dynamically allocated (which is less efficient in memory usage) and each item in the list also must store an additional pointer.