CS 2263: Systems Software Development

Jong-Kyou Kim, PhD

2022-10-24

Review

- struct is a collection of data
- New address space is allocated using `malloc` function
- self-referencing data structure enables implementing noble data structures such as the linked list and the binary tree
- These noble data structures use self-referencing and malloc/free

Administrivia

- Midterm: (tentatively) Nov 4 (Fri) at 3:30 pm (about 60 minutes)
  - Midterm exercise questions will be posted on Oct 31

Review: Sorting

```
int a[] = { 9,1,5,3,2,8,7,6,4 };
mysort(a);
for (int i = 0; i < 9; i++) {
  printf("%d\n", a[i]);
}
```

Remember?

```
int strcmp(const char* s1, const char* s2);
```

- s1 $<$ s2 $\longrightarrow$ negative
- s1 == s2 $\longrightarrow$ 0
- s1 $>$ s2 $\longrightarrow$ positive

A comparison function

- What does the following function do?

```
int cmp(const void* a, const void* b) {
        int* p = (int*)a;
        int* q = (int*)b;
        return *p - *q;
}
```

qsort

```
void main() {
        compare y = cmp2;
        int x[] = { 0,2,5,3,4 };
        qsort(x, 5, sizeof(int), y);
        for (int i = 0; i < 5; i++) {
                printf("%d\n", x[i]);
        }
}
```

qsort: struct Student

```
typedef struct Student {
  char name[30];
  float score;
} Student;

Student s[] = { {"Kim", 90}, ... };
```

- How many elements in the array `s`?
  - `sizeof(s)/sizeof(Student))`
- How to sort the given array by score?

  ⟶ a comparator function

qsort: struct Student

```
int cmp(const void* a, const void* b) {
        struct Student* p = (struct Student*)a;
        struct Student* q = (Struct Student*)b;
        return p->score - q->score;
}
```

struct Sorting

```
struct Score s[] = { {"Kim", 9}, ... };
mysort(a);
for (int i = 0; i < 9; i++) {
  printf("%d\n", a[i]);
}
```

⟶ By using a comparator function, `qsort()` can sort any type of array assuming that all the elements are of the same type

Remember? Doubly Linked list

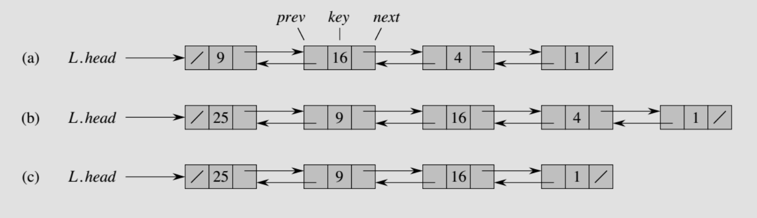


Figure: Concept of doubly linked list

Doubly Linked list

```
struct Node {
  int key;
  struct Node* prev;
  struct Node* next;
};
struct List {
  struct Node* head;
};
```

Doubly linked list

```
#include <stdio.h>
#include <malloc.h>

typedef struct _Node {
  int key;
  struct _Node* prev;
  struct _Node* next;
} Node;
typedef Node* pNode;
typedef struct {
  pNode head;
} List;
typedef List* pList;
```

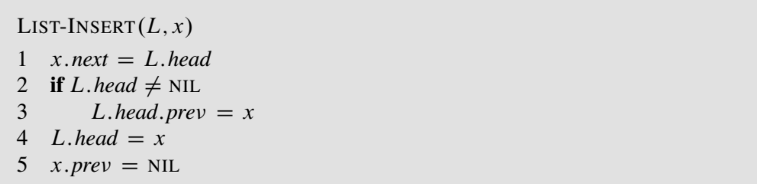
Allocating a Node

```
pNode list_new_node(int key) {
  pNode x = (pNode)malloc(sizeof(Node));
  x->key = key;
  return x;
}
```

Initialize the data structure

```
void list_init(pList plst) {
  plst->head = NULL;
}
```

Insert



```
LIST-INSERT$(L, x)$
$x.next = L.head$
if $L.head \neq$ NIL
    $L.head.prev = x$
$L.head = x$
$x.prev =$ NIL
```

Figure: Inserting a node

Insert implementation

```
void list_insert(pList plst, pNode x) {
  x->next = plst->head;
  if (plst->head != NULL) {
    plst->head->prev = x;
  }
  plst->head = x;
}
```

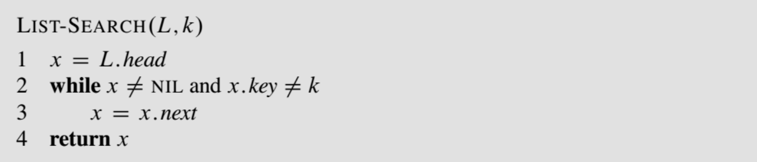
Debugging

```
void list_print(pList plst) {
  pNode x = plst->head;
  while(x) {
    printf("%d\n", x->key);
    x = x->next;
  }
}
```

Testing insert

```
void main()
{
  List lst;
  list_init(&lst);
  pNode x = list_new_node(1);
  list_insert(&lst, x);
  x = list_new_node(2);
  list_insert(&lst, x);
  x = list_new_node(3);
  list_insert(&lst, x);
  list_print(&lst);
}
```

Search



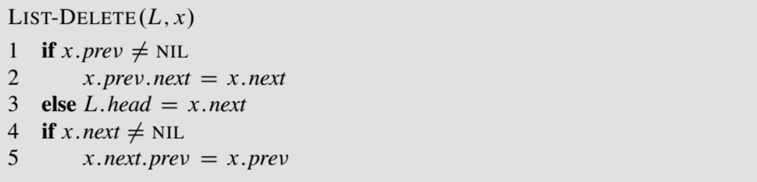
```
LIST-SEARCH$(L, k)$
$x = L.head$
while $x \neq$ NIL and $x.key \neq k$
    $x = x.next$
return $x$
```

Figure: Searching a value

Search

```
pNode list_search(pList plst, int key) {
  pNode x = plst->head;
  while(x && x->key != key) {
    x = x->next;
  }
  return x;
}
```

Delete



```
LIST-DELETE$(L, x)$
if $x.prev \neq$ NIL
    $x.prev.next = x.next$
else $L.head = x.next$
if $x.next \neq$ NIL
    $x.next.prev = x.prev$
```

Figure: Deleting a node

Delete

```
pNode list_delete(pList plst, pNode x) {
  if (x->prev != NULL) {
    x->prev->next = x->next;
  }
  else {
    plst->head = x->next;
  }
  if (x->next != NULL) {
    x->next->prev = x->prev;
  }
  free(x);
}
```

⟶ What is `free()`

- Return the allocated memory to the operating system so that other programs can use the memory

memory management in C

- Memory allocation: explicit call to `malloc()`
- Memory deallocation: explicit call to `free()`

⟶ How does the function free know how much memory to return?

  - It's automatically handled by the library implementation

Wrap-up

- We learned how to use self-referencing pointers to implement advanced data structures such as Doubly-Linked List
- Assuming the how the memory is organized, pointers enable general interfaces that are not limited by a specific type of arrays
- qsort is an example that uses the most general pointer `void*` and pointer to a function that compare two elements
- C uses malloc/free pair to manage memory dynamically

⟶ How much memory can malloc allocate?