Lecture 19: Linked List
Thomas Moon

March 31, 2022
• MP8 due this Thursday.
• Quiz4 is going on.

• Midterm2: 7pm, 4/7/2022. lec7-17
  • https://wiki.illinois.edu/wiki/display/ece220/Exams
  • Conflict: 5:30pm, 4/7/2022
  • Conflict sign-up by 4/3/2022
  • HKN review session: Sunday, 4/3, 3-5pm in ECEB1015
What if we need > 100 planes?
→ Increase the size of array.

But, sometimes we only need 2-3 planes
→ Wasting the rest of unused memory.

Planes take off & land, i.e. the data coming in & going out.
Adding or removing an item in the middle.
→ Not efficient in array

Dynamic Memory Allocation
(this lecture)

Linked List (next lecture)
Array
(can be automatic or dynamic)

Linked List
(dynamic only)

Memory allocated for array grid

Head pointer

Node 0
Node 1
Node 2
Node 3
NULL
A **linked list** is an ordered collection of **nodes**, each of which contains some data, connected using **pointers**.

- The first node in the list is called the **head**.
- The last node in the list is called the **tail**.

Each node contains at least
- a piece of data
- pointer to the next node in the list
Array vs. Linked List

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Allocation</td>
<td>Automatic/Dynamic</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Memory Structure</td>
<td>Contiguous</td>
<td>Not necessary consecutive</td>
</tr>
<tr>
<td>Order of Access</td>
<td>Random</td>
<td>Sequential</td>
</tr>
<tr>
<td>Insertion/Deletion</td>
<td>Create/delete space, then shift all successive elements</td>
<td>Change pointer address</td>
</tr>
</tbody>
</table>
Example: Student Record

typedef struct StudentStruct{
    int UIN;
    char *netid;
    float GPA;
} student;

typedef struct StudentStruct{
    int UIN;
    char *netid;
    float GPA;
    struct StudentStruct *next;
} node;
Task 1: Insert a new node at the head
Recall 1: Solve Swap Problem

```c
void Swap(int firstVal, int secondVal);
int main()
{
    int valueA = 1;
    int valueB = 2;

    Swap(valueA, valueB);
}
```

**call by value**

```c
void NewSwap(int *firstVal, int *secondVal);
int main()
{
    int valueA = 1;
    (int valueB = 2;

    NewSwap(&valueA, &valueB);
}
```

**call by reference**

```c
void NewSwap(int *firstVal, int *secondVal)
{
    int tempVal;

    tempVal = *firstVal;
    *firstVal = *secondVal;
    *secondVal = tempVal;
}
```
Recall2: Double Pointer

```c
int x = 10;
int *p;
p = &x;

int **pp;
pp = &p;

printf("x%X\n", &pp);
printf("x%X\n", pp);
printf("x%X\n", *pp);
printf("%d\n", **pp);
```
Recall 3: Pass Structs as Arguments

A. Passing by value will push the entire struct members onto the run-time stack.
B. Pass only one pointer.

void print_flightName(Flight plane)
{
    printf("flight name: %s\n", plane.flightName);
}

void print_flightName(Flight *plane)
{
    printf("flight name: %s\n", plane->flightName);
}

vs

which one is more efficient?
Task1: Insert a new node at the head

Run-time stack

```c
void insert_head
```

Heap

Linked list is empty

```c
int main
```

```c
void insert_head(){
}
```
Task1: Insert a new node at the head

```c
void insert_head()
{
    node *temp = (node*) malloc(sizeof(node));
}
```
Task1: Insert a new node at the head

```c
void insert_head(node **headpptr){
    node *temp = (node*) malloc(sizeof(node));
    *headpptr = temp;
}
```
Task 1: Insert a new node at the head

```c
void insert_head
```

```c
int main
```

```c
void insert_head(node **headpptr, node *data){
    node *temp = (node*) malloc(sizeof(node));
    *temp = *data;
    *headpptr = temp;
}
```
Task1: Insert a new node at the head

Run-time stack

```c
void insert_head

int main

void insert_head(node **headpptr, node *data){
    node *temp = (node*) malloc(sizeof(node));
    *temp = *data;
    temp->next = *headpptr;
    *headpptr = temp;
}
```
Task 1: Insert a new node at the head

Run-time stack

```c
void insert_head
```

Heap

```c
int main
```

After tear-down activation record for `insert_head`. 

```c
void insert_head(node **headpptr, node *data){
    node *temp = (node*) malloc(sizeof(node));
    *temp = *data;
    temp->next = *headpptr;
    *headpptr = temp;
}
```
void insert_head(node **headpptr, node *data);

int main() {
    node *headptr = NULL;
    node data;
    data.UIN = 0;
    ...

    insert_head(&headptr, &data);
}

double pointer because we need to modify the head pointer in main
void printStudents(node *cursor){
    printf("UIN netid GPA\n");
    while(cursor != NULL){
        printf("%d %s %f\n", cursor->UIN, cursor->netid, cursor->GPA);
        cursor = cursor->next;
    }
}

typedef struct StudentStruct{
    int UIN;
    char *netid;
    float GPA;
    struct StudentStruct *next;
}node;
Task 3: Insert a new node in a sorted way (GPA, descending)

```c
void insert_sorted_GPA(node **headpptr, node *data){
    node *temp = (node*) malloc(sizeof(node));
    *temp = *data;
}
```

```c
int main()
{
    head ptr
    data
}
```
void insert_sorted_GPA(node **headpptr, node *data){
    node *temp = (node*) malloc(sizeof(node));
    *temp = *data;
    while((*(headpptr))->GPA > temp->GPA)
    {
        //traverse
        headpptr = (*(headpptr))->next;
    }
    temp->next = *headpptr;
}
Task3: Insert a new node in a sorted way (GPA, descending)

Run-time stack

```
void insert_sorted_GPA()
```

Heap

```
void insert_sorted_GPA(node **headpptr, node *data){
    node *temp = (node*) malloc(sizeof(node));
    *temp = *data;
    while( (*headpptr)-GPA > temp->GPA ){
        //traverse
        headpptr = (*headpptr)->next;
    }
    temp->next = *headpptr;
    *headpptr = temp;
}
```

int main()

```
Task3: Insert a new node in a sorted way (GPA, descending)

```c
void insert_sorted_GPA(void *headpptr, node *data){
    node *temp = (node*) malloc(sizeof(node));
    *temp = *data;
    while( (*headpptr)->GPA > temp->GPA && *headpptr != NULL){
        //traverse
        headpptr = (*headpptr)->next;
    }
    temp->next = *headpptr;
    *headpptr = temp;
}
```
Task 3: Insert a new node in a sorted way (GPA, descending)

```c
void insert_sorted_GPA(node **headpptr, node *data)
{
    node *temp = (node*) malloc(sizeof(node));
    *temp = *data;
    while( (*headpptr)->GPA > temp->GPA && *headpptr != NULL)
    {
        //traverse
        headpptr = (*headpptr)->next;
    }
    temp->next = *headpptr;
    *headpptr = temp;
}
```

void insert_sorted_GPA()

Run-time stack

Heap
Task 4: Delete all nodes

Run-time stack

Heap

```c
void delete_all(){
}
```