Umut Oztok

July 1, 2016
Arrays

- A basic data structure (commonly used).
- Organize data in a sequential way.
Arrays

- A basic data structure (commonly used).
- Organize data in a sequential way.
- Pros:

```plaintext
```
Arrays

- A basic data structure (commonly used).
- Organize data in a sequential way.
- Pros:
  - Easy access to elements, i.e., array[5] gets 6th element.
  - Easy to debug.
- Cons:
Arrays

A basic data structure (commonly used).
Organize data in a sequential way.
Pros:
  - Easy access to elements, i.e., array[5] gets $6^{th}$ element.
  - Easy to debug.
Cons:
  - Fixed size.
  - Not easy to insert new elements to the front/middle.
Another data structure.
Like arrays, organizes data in a sequential way.
In general, linked lists are strong when arrays are weak, and vice versa.
- Another data structure.
- Like arrays, organizes data in a sequential way.
- In general, linked lists are strong when arrays are weak, and vice versa.
- Basic component is a node:

```
struct Node {
    DataType value;
    Node* next;
};
```
Another data structure.
Like arrays, organizes data in a sequential way.
In general, linked lists are strong when arrays are weak, and vice versa.
Basic component is a node:

```
struct Node {
    dataType value;
    Node* next;
};
```

Self-referential structures:

```
typedef int dataType;
struct Node {
    dataType value;
    Node* next;
};
```
A linked list is a series of nodes, each pointing to next one.

- Head pointer to point the first element.
- Last node’s next pointer is NULL.
Linked Lists

Some operations:

- **Insert:**
  - To the front.
  - To the end.
  - In the middle.

- **Delete.**

- **Search.**

- **Traverse.**
How to traverse the nodes of a linked list?

```cpp
void print() {
    Node* cur = head;
    while (cur != nullptr) {
        cout << cur->value << " ";
        cur = cur->next;
    }
}
```

Output: 1 5 3
How to traverse the nodes of a linked list?

```cpp
void print() {
    Node* cur = head;
    while (cur != nullptr) {
        cout << cur->value << " ";
        cur = cur->next;
    }
}
```

Output: 1 5 3
How to traverse the nodes of a linked list?

```cpp
void print() {
    Node* cur = head;
    while (cur != nullptr) {
        cout << cur->value << " ";
        cur = cur->next;
    }
}
```

Output: 1 5 3
How to traverse the nodes of a linked list?

```cpp
void print() {
    Node* cur = head;
    while (cur != nullptr) {
        cout << cur->value << " ";
        cur = cur->next;
    }
}
```

Output: 1 5 3
How to traverse the nodes of a linked list?

```
void print() {
    Node* cur = head;
    while (cur != nullptr) {
        cout << cur->value << " ";
        cur = cur->next;
    }
}
```

Output: 1 5 3
How to traverse the nodes of a linked list?

```cpp
void print() {
    Node* cur = head;
    while (cur != nullptr) {
        cout << cur->value << " ";
        cur = cur->next;
    }
}
```

Output: 1 5 3
How to traverse the nodes of a linked list?

```
void print() {
    Node* cur = head;
    while(cur != nullptr) {
        cout << cur->value << " ";
        cur = cur->next;
    }
}
```

Output: 1 5 3
How to insert a node with value 6 to the front of the list?

**From:**

```
head → 1 → 5 → 3 → Nil
```

**To:**

```
head → 6 → 1 → 5 → 3 → Nil
```

**Steps:**

```
void insertToFront(int val) {
    Node* p = new Node;
    p->value = val;
    p->next = head;
    head = p;
}
```
How to insert a node with value 6 to the front of the list?

From:

```
head -> 1  -> 5  -> 3  -> Nil
```

To:

```
head -> 6  -> 1  -> 5  -> 3  -> Nil
```

Steps:
- Create a new node.
- Point the node’s next pointer to head.
- Point head to the node.
How to insert a node with value 6 to the front of the list?

**From:**

```
head -> 1 -> 5 -> 3 -> Nil
```

**To:**

```
head -> 6 -> 1 -> 5 -> 3 -> Nil
```

**Steps:**
- Create a new node.
- Point the node’s next pointer to head.
- Point head to the node.

```c
void insertToHead(int val) {
    Node* p = new Node;
    p->value = val;
    p->next = head;
    head = p;
}
```
How to insert a node with value 6 to the end of the list?

**From:**

```
head → 1 → 5 → 3 → Nil
```

**To:**

```
head → 1 → 5 → 3 → 6 → Nil
```

**Steps:**

1. Create a new node.
2. Go to the end of the list.
3. Point last node’s next pointer to the new node.
How to insert a node with value 6 to the end of the list?

**From:**

```
head -> 1 -> 5 -> 3 -> Nil
```

**To:**

```
head -> 1 -> 5 -> 3 -> 6 -> Nil
```

**Steps:**

- Create a new node.
- Go to the end of the list.
- Point last node’s next pointer to the new node.
```c
void insertToEnd(int val) {
    Node* p = new Node;
    p->value = val;
    p->next = nullptr;
    Node* cur = head;
    if (cur == nullptr) head = p;
    else {
        while (cur->next != nullptr)
            cur = cur->next;
        cur->next = p;
    }
}
```
void insertToEnd(int val) {
    Node* p = new Node;
    p->value = val;
    p->next = nullptr;

    Node* cur = head;
    if (cur == nullptr) head = p;
    else {
        while (cur->next != nullptr)
            cur = cur->next;
        cur->next = p;
    }
}
```cpp
void insertToEnd(int val) {
    Node* p = new Node;
    p->value = val;
    p->next = nullptr;

    Node* cur = head;
    if (cur == nullptr) head = p;
    else {
        while (cur->next != nullptr)
            cur = cur->next;
        cur->next = p;
    }
}
```
Insertion [To the $K^{th}$ slot]

How to insert a node with value 6 to the $1^{st}$ slot? (Node with value 1 is located at the $0^{th}$ slot.)

**From:**

```
head → 1 → 5 → 3 → Nil
```

**To:**

```
head → 1 → 6 → 5 → 3 → Nil
```

**Steps:**

1. Find the node $N$ at $K-1$ slot.
2. Create a new node.
3. "Add" the node to $K$th slot. That is:
   - Make new node's next pointer point to the node at $K$th slot.
   - Make $N$'s next pointer point to the new node.
4. What if $K=0$?
How to insert a node with value 6 to the 1st slot? (Node with value 1 is located at the 0th slot.)

**From:**

\[
\text{head} \rightarrow 1 \rightarrow 5 \rightarrow 3 \rightarrow \text{Nil}
\]

**To:**

\[
\text{head} \rightarrow 1 \rightarrow 6 \rightarrow 5 \rightarrow 3 \rightarrow \text{Nil}
\]

**Steps:**

- Find the node \(N\) at \(K - 1\)st slot.
- Create a new node.
- "Add" the node to \(K\)th slot. That is:
  - Make new node’s next pointer point to the node at \(K\)th slot.
  - Make \(N\)’s next pointer point to the new node.
How to insert a node with value 6 to the 1<sup>st</sup> slot? (Node with value 1 is located at the 0<sup>th</sup> slot.)

From:

```
head → 1 → 5 → 3 → Nil
```

To:

```
head → 1 → 6 → 5 → 3 → Nil
```

Steps:

- Find the node \( N \) at \( K - 1^{st} \) slot.
- Create a new node.
- "Add" the node to \( K^{th} \) slot. That is:
  - Make new node’s next pointer point to the node at \( K^{th} \) slot.
  - Make \( N \)’s next pointer point to the new node.
- What if \( K = 0 \)?
void insertToKth(int val, int index) {
  if (head == nullptr || index == 0) insertToFront(val);
  else {
    Node* cur = head;
    while (cur->next != nullptr) {
      if (--index == 0) break;
      cur = cur->next;
    }
    //cur points to either the last element
    //or (index-1)st element
    Node* p = new Node;
    p->value = val;
    p->next = cur->next;
    cur->next = p;
  }
}
void insertToKth(int val, int index) {
  if (head == nullptr || index == 0) insertToFront(val);
  else {
    Node* cur = head;
    while (cur->next != nullptr) {
      if (--index == 0) break;
      cur = cur->next;
    }
    //cur points to either the last element
    //or (index-1)st element
    Node* p = new Node;
    p->value = val;
    p->next = cur->next;
    cur->next = p;
  }
}
void insertToKth(int val, int index) {
    if (head == nullptr || index == 0) insertToFront(val);
    else {
        Node* cur = head;
        while (cur->next != nullptr) {
            if (--index == 0) break;
            cur = cur->next;
        }
        //cur points to either the last element or (index-1)st element
        Node* p = new Node;
        p->value = val;
        p->next = cur->next;
        cur->next = p;
    }
}
void insertToKth(int val, int index) {
    if (head == nullptr || index == 0) insertToFront(val);
    else {
        Node* cur = head;
        while (cur->next != nullptr) {
            if (--index == 0) break;
            cur = cur->next;
        }
        // cur points to either the last element
        // or (index-1)st element
        Node* p = new Node;
        p->value = val;
        p->next = cur->next;
        cur->next = p;
    }
}
How to delete the first node containing 5?

From:

head → 1 → 5 → 3 Nil

To:

head → 1 → 3 Nil

Steps: (Let $N$ denote the node we want to delete.)
Deletion

How to delete the first node containing 5?
From:

\[
\text{head} \rightarrow 1 \rightarrow 5 \rightarrow 3 \rightarrow \text{Nil}
\]

To:

\[
\text{head} \rightarrow 1 \rightarrow 3 \rightarrow \text{Nil}
\]

Steps: (Let \( N \) denote the node we want to delete.)

- Find the node just before \( N \).
- Have its next pointer point to the node just after \( N \).
- Delete \( N \).
- Watch out for special cases.
  - List is empty.
  - List contains a single node.
  - List doesn’t have a node containing the given value.
Deletion

```c
void delete(int val) {
```
Deletion

void delete(int val) {
    if (head == nullptr) return;
}

if (head == nullptr) return;

Node* p = head;
head = p->next;
delete p;

else {
    Node* cur = head;
    while (cur->next != nullptr) {
        if (cur->next->value == val) break;
        cur = cur->next;
    }
    if (cur->next == nullptr) return;
    Node* p = cur->next;
    cur->next = p->next;
    delete p;
}
void delete(int val) {
    if (head == nullptr) return;
    if (head->value == val) {
        Node* p = head;
        head = p->next;
        delete p;
    }
    else {
}
```c
void delete(int val) {
    if(head == nullptr) return;
    if(head->value == val) {
        Node* p = head;
        head = p->next;
        delete p;
    }
    else {
        Node* cur = head;
        while(cur->next != nullptr) {
            if(cur->next->value == val) break;
            cur = cur->next;
        }
        if(cur->next == nullptr) return;
        Node* p = cur->next;
        cur->next = p->next;
        delete p;
    }
}
```
```cpp
void delete(int val) {
    if (head == nullptr) return;
    if (head->value == val) {
        Node* p = head;
        head = p->next;
        delete p;
    }
    else {
        Node* cur = head;
        while (cur->next != nullptr) {
            if (cur->next->value == val) break;
            cur = cur->next;
        }
        if (cur->next == nullptr) return;
        Node* p = cur->next;
        cur->next = p->next;
        delete p;
    }
}
```
Exercise 1

Append one linked list to another.
Given two linked lists:

- `head`: 1 → 5 → 3 → Nil
- `head2`: 8 → 4 → Nil

Append the second one to the first one:

- `head`: 1 → 5 → 3 → 8 → 4 → Nil
Exercise 1

Append one linked list to another.
Given two linked lists:

head
1 → 5 → 3 → Nil

head2
8 → 4 → Nil

Append the second one to the first one:

head
1 → 5 → 3 → 8 → 4 → Nil

Any volunteers?

void append(Node* head2) {
    ...
}

Umut Oztok  CS32 - Week 2
void append(Node* head2) {
  if (head == nullptr) head = head2;
  Node* cur = head;
  while (cur->next != nullptr) cur = cur->next;
  // cur pointing to the last node of the first list
  cur->next = head2;
}
Exercise 1

```cpp
void append(Node* head2) {
    if (head == nullptr) head = head2;
    Node* cur = head;
    while (cur->next != nullptr) cur = cur->next;
    //cur pointing to the last node of the first list
    cur->next = head2;
}
```
Exercise 1

```c
void append(Node* head2) {
    if(head == nullptr) head = head2;

    Node* cur = head;
```
void append(Node* head2) {
    if(head == nullptr) head = head2;

    Node* cur = head;
    while(cur->next != nullptr) cur = cur->next;
    cur->next = head2;
}
void append(Node* head2) {
    if (head == nullptr) head = head2;

    Node* cur = head;
    while (cur->next != nullptr) cur = cur->next;
    //cur pointing to the last node of the first list
    cur->next = head2;
}

Reverse a linked list.

From:

To:
Exercise 2

Reverse a linked list.

**From:**

```
head -> 1 -> 5 -> 3 -> Nil
```

**To:**

```
head -> 3 -> 5 -> 1 -> Nil
```

Any volunteers?

```java
void reverse() {
    ...
}
```
void reverse() {
    // Implementation of reversing a linked list.
}
void reverse() {
    if (head == nullptr) return;
    Node* p = head;
    Node* q = p->next;
    p->next = NULL;
    while (q != NULL) {
        Node* r = q->next;
        q->next = p;
        p = q;
        q = r;
    }
    head = p;
}
void reverse() {
    if (head == nullptr) return;
    if (head->next == nullptr) return;
    Node* p = head;
    Node* q = p->next;
    p->next = NULL;
    while (q != NULL) {
        Node* r = q->next;
        q->next = p;
        p = q;
        q = r;
    }
    head = p;
}
void reverse() {
    if (head == nullptr) return;
    if (head->next == nullptr) return;

    Node* p = head;
    Node* q = p->next;

    p->next = NULL;
    while (q != NULL) {
        Node* r = q->next;
        q->next = p;
        p = q;
        q = r;
    }
    head = p;
}
void reverse() {
    if (head == nullptr) return;
    if (head->next == nullptr) return;

    Node* p = head;
    Node* q = p->next;
    p->next = NULL;
    p->next = NULL;
    Node* r = q->next;
    q->next = p;
    p = q;
    q = r;
}
}
void reverse() {
    if (head == nullptr) return;
    if (head->next == nullptr) return;

    Node* p = head;
    Node* q = p->next;
    p->next = NULL;
    while (q != NULL) {
        Node* r = q->next;
        q->next = p;
        p = q;
        q = r;
    }
    head = p;
}
```c
void reverse() {
    if (head == nullptr) return;
    if (head->next == nullptr) return;

    Node* p = head;
    Node* q = p->next;
    p->next = NULL;
    while (q != NULL) {
        Node* r = q->next;
        q->next = p;
        p = q;
        q = r;
    }

    head = p;
}
```
Things to be careful about while playing with linked lists:
- Nullity of a pointer before deref’ing it.
- Empty linked lists.
- Linked lists with one element.
To insert an element at the end of a list, you need to go over each node, which may be quite inefficient.

If you always keep a pointer pointing to the last element, inserting an element at the end can be done immediately.

That’s what a "tail pointer" is.

Note that you have to maintain tail pointer in most functions.
Linked lists are asymmetric in the sense that you can traverse the list in only one direction.

Doubly linked lists allow you to traverse in both directions. Basically, every node keeps track of the elements coming just before and after itself.

As you have to deal with more pointers, it is harder to maintain.
A method to simplify your code.
Create a **dummy** node when you initialize a linked list.
Keep the dummy node until destructing the linked list.
So, the head pointer always points to the dummy node.
No special case checking for empty linked lists.
Be careful, the first actual element is head->next.
Linked List Properties

- Dynamic allocated memory - flexible.
  - Comes with a cost?
Linked List Properties

- Dynamic allocated memory - flexible.
  - Comes with a cost? Memory of pointers.
Linked List Properties

- Dynamic allocated memory - flexible.
  - Comes with a cost? Memory of pointers.
- Fast insertion (to the front or in the middle).
Linked List Properties

- Dynamic allocated memory - flexible.
  - Comes with a cost? Memory of pointers.
- Fast insertion (to the front or in the middle).
  - Also to the end with tail pointers.
Dynamic allocated memory - flexible.
  - Comes with a cost? Memory of pointers.
Fast insertion (to the front or in the middle).
  - Also to the end with tail pointers.
Fast deletion (no need to shift elements compared to array).
Linked List Properties

- Dynamic allocated memory - flexible.
  - Comes with a cost? Memory of pointers.
- Fast insertion (to the front or in the middle).
  - Also to the end with tail pointers.
- Fast deletion (no need to shift elements compared to array).
- Search is almost the same as array.
Linked List Properties

- Dynamic allocated memory - flexible.
  - Comes with a cost? Memory of pointers.
- Fast insertion (to the front or in the middle).
  - Also to the end with tail pointers.
- Fast deletion (no need to shift elements compared to array).
- Search is almost the same as array.
- Random access is the weakest point of linked lists.
Slides will be available at
http://www.cs.ucla.edu/~umut/cs32