MTH 4300/4299: Programming and Computer Science II

Lecture 06: Linked lists

Jaime Abbariao

Updates from last class

Why was it okay to have negative indices when I initially said it wasn't?

```cpp
#include <vector>
int main() {
  std::vector<int> nums = {1, 2, 3, 4, 5};
  int *p = nums + 1;

  // Why is this okay?
  std::cout << p[-1] << std::endl;
  return 0;
}
```

`p[i]` is just shorthand for `*(p + i)`. So `p[-1]` means `*(p + (-1))`, which is `*(p - 1)`.

Since `p` points to `nums[1]` (the second element), `p - 1` points to `nums[0]` (the first element). That's a perfectly valid memory location within the array.

The rule isn't "negative indices are always illegal" -- the rule is you must not access memory outside the bounds of the array. If `p` points somewhere in the middle of an array, then `p[-1]` is fine because it still lands inside the array.

What would be undefined behavior:

```cpp
int arr[] = {1, 2, 3};
int* p = arr;       // p points to arr[0], the very start
std::cout << p[-1]; // undefined behavior! goes before the array
```

Structs

A `struct` is a way to group related variables together under one name.

```cpp
struct Student {
    std::string name;
    int id;
    double gpa;
};

int main() {
    Student s;
    s.name = "Alice";
    s.id = 12345;
    s.gpa = 3.8;

    std::cout << s.name << " (ID: " << s.id
              << ") GPA: " << s.gpa << std::endl;
    return 0;
}
```

Think of a struct as a custom type that bundles data together. Each piece of data inside is called a member or field.

Structs and pointers

When you have a pointer to a struct, use the arrow operator `->` to access members:

```
Student s;
s.name = "Bob";
s.id = 67890;
s.gpa = 3.5;

Student* ptr = &s;

// These two lines are equivalent:
std::cout << (*ptr).name << std::endl;
std::cout << ptr->name << std::endl;
```

The `->` operator is shorthand for dereferencing and then accessing a member. You will use this constantly with linked lists.

▒▒ Constructors

A constructor is a special function that runs automatically when you create an instance of a struct. It lets you initialize the members right away.

```
struct Student {
    std::string name;
    int id;
    double gpa;

    // Constructor
    Student(std::string n, int i, double g) {
        name = n;
        id = i;
        gpa = g;
    }
};

int main() {
    Student s("Alice", 12345, 3.8);
    std::cout << s.name << std::endl;
    return 0;
}
```

▒▒ Default constructors

If you define a constructor with parameters, C++ no longer provides a default (no-argument) constructor. You can add one back explicitly:

```
struct Student {
    std::string name;
    int id;
    double gpa;

    // Default constructor
    Student() {
        name = "";
        id = 0;
        gpa = 0.0;
    }

    // Parameterized constructor
    Student(std::string n, int i, double g) {
        name = n;
        id = i;
        gpa = g;
    }
};

int main() {
    Student a;                          // uses default constructor
    Student b("Bob", 67890, 3.5);     // uses parameterized constructor
    return 0;
}
```

▒▒▒ Exercise: Structs

Define a struct `Point` with two `double` members: `x` and `y`.

1. Write a constructor that takes `x` and `y` as parameters
2. Write a function `double distance(Point* a, Point* b)` that computes the Euclidean distance between two points using the arrow operator
3. In `main`, create two points `(3.0, 0.0)` and `(0.0, 4.0)`, and print the distance between them

Hint: The distance formula is `sqrt((x2-x1)^2 + (y2-y1)^2)`. Use `#include <cmath>` for `std::sqrt`.

Expected output:

```
5
```

▒▒ Linked Lists

An array stores elements in contiguous memory. A linked list stores elements in separate nodes, where each node points to the next one.

```
Array:       [10][20][30][40][50]    (contiguous in memory)

Linked list: [10|*]->[20|*]->[30|*]->[40|*]->[50|nullptr]
              head
```

Why use a linked list over an array?

- Inserting or deleting at the front is O(1) (arrays are O(n) because you need to shift elements)
- No need to know the size in advance or reallocate
- Easy to insert/delete in the middle if you already have a pointer to the location

Trade-offs:

- No random access (can't jump to element `i` directly)
- Extra memory per element (each node stores a pointer)
- Worse cache performance (nodes scattered in memory)

▒▒ The Node struct

The building block of a linked list is a node:

```
struct Node {
    int data;
    Node* next;

    Node(int val) {
        data = val;
        next = nullptr;
    }
};
```

Each node holds:

1. A piece of data (`data`)
2. A pointer to the next node (`next`), or `nullptr` if it's the last node

▒▒ Building a list by hand

```
int main() {
    Node* head = new Node(10);
    head->next = new Node(20);
    head->next->next = new Node(30);

    // head -> [10] -> [20] -> [30] -> nullptr
    return 0;
}
```

`new` allocates memory on the heap and returns a pointer. Unlike stack variables, heap memory persists until you explicitly free it with `delete`.

We'll talk more about `new`/`delete` and memory management in a future lecture.

Common operations

Iteration

To visit every node, start at `head` and follow the `next` pointers:

```
void print_list(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        std::cout << current->data << " -> ";
        current = current->next;
    }
    std::cout << "nullptr" << std::endl;
}
```

```
Output: 10 -> 20 -> 30 -> nullptr
```

This pattern -- walking through the list with a `current` pointer -- is the foundation for almost every linked list operation.

Getting the length

```
int length(Node* head) {
    int count = 0;
    Node* current = head;
    while (current != nullptr) {
        count++;
        current = current->next;
    }
    return count;
}
```

Unlike arrays, you can't get the length in O(1) -- you must traverse the entire list.

▒▒▒ Exercise: Linked list search

Write a function `bool contains(Node* head, int target)` that returns `true` if any node in the list has `data` equal to `target`, and `false` otherwise.

Test it in `main` by building the list `10 -> 20 -> 30 -> nullptr` and searching for `20` (should print `true`) and `50` (should print `false`).

Hint: Use the same traversal pattern as `print_list`.

```cpp
bool contains(Node* head, int target) {
    Node* current = head;
    while (current != nullptr) {
        if (current->data == target) {
            return true;
        }
        current = current->next;
    }
    return false;
}

int main() {
    Node* head = new Node(10);
    head->next = new Node(20);
    head->next->next = new Node(30);

    std::cout << std::boolalpha;
    std::cout << contains(head, 20) << std::endl;  // true
    std::cout << contains(head, 50) << std::endl;  // false
    return 0;
}
```

▒▒▒ Insertion

▒▒▒▒ Inserting at the front (prepend)

To insert at the front, create a new node and point it to the current head:

```
void prepend(Node*& head, int val) {
    Node* new_node = new Node(val);
    new_node->next = head;
    head = new_node;
}
```

```
Before: head -> [10] -> [20] -> [30] -> nullptr
After:  head -> [5] -> [10] -> [20] -> [30] -> nullptr
```

Notice the parameter is `Node*& head` -- a reference to a pointer. This lets the function update the caller's `head` variable.

▒▒ Insertion (continued)

▒▒ Inserting at the back (append)

To insert at the end, walk to the last node, then attach a new node:

```
void append(Node*& head, int val) {
    Node* new_node = new Node(val);
    if (head == nullptr) {
        head = new_node;
        return;
    }
    Node* current = head;
    while (current->next != nullptr) {
        current = current->next;
    }
    current->next = new_node;
}
```

Note the special case: if the list is empty (`head == nullptr`), the new node becomes the head.

Insertion (continued)

Inserting after a given node

```
void insert_after(Node* node, int val) {
    if (node == nullptr) {
        return;
    }
    Node* new_node = new Node(val);
    new_node->next = node->next;
    node->next = new_node;
}
```

```
Before: [10] -> [20] -> [30] -> nullptr
                  ^
                node

After:  [10] -> [20] -> [25] -> [30] -> nullptr
```

The order of pointer assignments matters! If you set `node->next = new_node` first, you lose the reference to the rest of the list.

▒▒▒ Exercise: Build a list with append

Using the `append` function above, write a `main` that:

1. Starts with an empty list (`Node* head = nullptr;`)
2. Appends the values 1, 2, 3, 4, 5 in a loop
3. Prints the list using `print_list`

Expected output:

```
1 -> 2 -> 3 -> 4 -> 5 -> nullptr
```

▒▒ Exercise: Build a list with append (Solution)

```
int main() {
    Node* head = nullptr;
    for (int i = 1; i <= 5; i++) {
        append(head, i);
    }
    print_list(head);
    return 0;
}
```

Deletion

Deleting the front node

```
void delete_front(Node*& head) {
    if (head == nullptr) {
        return;
    }
    Node* temp = head;
    head = head->next;
    delete temp;
}
```

We save the old head in `temp` so we can `delete` it after moving the head pointer forward. If you just did `head = head->next`, the old node would be leaked (still allocated but unreachable).

Deletion (continued)

Deleting a node by value

```
void delete_value(Node*& head, int target) {
    if (head == nullptr) {
        return;
    }

    // Special case: target is at the head
    if (head->data == target) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

    // General case: find the node before the target
    Node* current = head;
    while (current->next != nullptr) {
        if (current->next->data == target) {
            Node* temp = current->next;
            current->next = temp->next;
            delete temp;
            return;
        }
        current = current->next;
    }
}
```

Key idea: to remove a node, you need a pointer to the node before it, so you can "skip over" it by updating `next`.

▒▒ Deletion illustrated

```
Deleting node with value 20:

Step 1: Find the node before the target
  [10] -> [20] -> [30] -> nullptr
    ^
  current   (current->next->data == 20)

Step 2: Save pointer to target, skip over it
  temp = current->next         // temp points to [20]
  current->next = temp->next   // [10] now points to [30]

Step 3: Free the memory
  delete temp

Result: [10] -> [30] -> nullptr
```

▒▒▒ Exercise: Delete all occurrences

Write a function `void delete_all(Node*& head, int target)` that removes all nodes with the given value from the list, not just the first one.

Test with the list `1 -> 2 -> 3 -> 2 -> 4 -> 2 -> nullptr` and delete all `2`s.

Expected output:

```
1 -> 3 -> 4 -> nullptr
```

Hint: Be careful when the head itself has the target value (it might happen multiple times).

```cpp
void delete_all(Node*& head, int target) {
    // Handle target values at the head
    while (head != nullptr && head->data == target) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }

    if (head == nullptr) {
        return;
    }

    // Handle target values in the rest of the list
    Node* current = head;
    while (current->next != nullptr) {
        if (current->next->data == target) {
            Node* temp = current->next;
            current->next = temp->next;
            delete temp;
        } else {
            current = current->next;
        }
    }
}

int main() {
    Node* head = nullptr;
    int vals[] = {1, 2, 3, 2, 4, 2};
    for (int i = 5; i >= 0; i--) {
        prepend(head, vals[i]);
    }
    delete_all(head, 2);
    print_list(head);
    return 0;
}
```

▒▒ Common problems

These are classic linked list problems you should be comfortable with:

1. Reverse a linked list

```
Node* reverse(Node* head) {
    Node* prev = nullptr;
    Node* current = head;
    while (current != nullptr) {
        Node* next_node = current->next;
        current->next = prev;
        prev = current;
        current = next_node;
    }
    return prev;
}
```

Walk through the list, flipping each `next` pointer to point backwards. You need three pointers: `prev`, `current`, and `next_node`.

▒▒ Common problems (continued)

2. Find the middle node

Use the "slow and fast pointer" technique: move one pointer by 1 step and another by 2 steps. When the fast pointer reaches the end, the slow pointer is at the middle.

```
Node* find_middle(Node* head) {
    Node* slow = head;
    Node* fast = head;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

3. Detect a cycle

The same slow/fast technique can detect cycles: if the list has a cycle, the fast pointer will eventually "lap" the slow pointer and they'll meet.

```
bool has_cycle(Node* head) {
    Node* slow = head;
    Node* fast = head;
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) {
            return true;
        }
    }
    return false;
}
```

▒▒▒ Lab

Write a complete program that implements a linked list with the following functions:

1. `void prepend(Node*& head, int val)` -- insert at the front
2. `void append(Node*& head, int val)` -- insert at the end
3. `void print_list(Node* head)` -- print the list in the format `val -> val -> ... -> nullptr`
4. `int length(Node* head)` -- return the number of nodes
5. `void delete_value(Node*& head, int target)` -- delete the first occurrence of a value
6. `Node* reverse(Node* head)` -- reverse the list and return the new head

Then in `main`:

- Build the list `1 -> 2 -> 3 -> 4 -> 5` using `append`
- Print the list and its length
- Delete the value `3` and print the list
- Reverse the list and print it

Expected output:

```
1 -> 2 -> 3 -> 4 -> 5 -> nullptr
Length: 5
1 -> 2 -> 4 -> 5 -> nullptr
5 -> 4 -> 2 -> 1 -> nullptr
```