Formal DSA in C++

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

Linked lists

1.1 Singly-linked lists

If a node contains a data member that is a pointer to another node, then many nodes can be strung together using only one variable to access the entire sequence of nodes. Such a sequence of nodes is the most frequently used implementation of a linked list, which is a data structure composed of nodes, each node holding some information and a pointer to another node in the list. If a node has a link only to its successor in this sequence, the list is called a singly linked list

Each node resides on the heap

Linked lists can easily grow and shrink in size without reallocating memory or moving elements. Adding or removing nodes (especially at the beginning or middle) is more efficient compared to arrays, as no shifting of elements is required. Memory is allocated as needed, avoiding wasted space typical in arrays with fixed sizes.

However, each node requires extra memory for the pointer to the next node. Accessing elements requires traversal from the head, making lookups slower (O(n)) compared to arrays, which offer O(1) access via indexing. Nodes are scattered in memory, leading to poor cache performance compared to arrays, which have contiguous memory locations.

1.1.1 Structure of the node

The node structure is typically implemented in the following way

```
struct node {
    node* next = nullptr;
    T data = 0;

    node() = default;
    node(data) : data(data) {}
    node(next, data) : next(next), data(data) {}
}
```

A node includes two data members: info and next. The info member is used to store information, and this member is important to the user. The next member is used to link nodes to form a linked list. It is an auxiliary data member used to maintain the linked list. It is indispensable for implementation of the linked list, but less important (if at all) from the user's perspective. Note that node is defined in terms of itself because one data member, next, is a pointer to a node of the same type that is just being defined. Objects that include such a data member are called self-referential objects.

1.1.2 The list class/struct

We also implement the list structure as a class or struct.

```
class single_list {
    node* head = nullptr;
public:
    ...
};
```

1.1.3 Interface of a singly linked list stack

The interface typically includes the following operations:

1. **Insert:** Add a node at the beginning, end, or a specific position in the list.
2. **Delete:** Remove a node from the beginning, end, or a specific position.
3. **Search:** Find a node with a given value.
4. **Traverse:** Iterate through the list to access or print each node's data.
5. **IsEmpty:** Check if the list is empty.
6. **Size:** Return the number of nodes in the list. The first node is called the head, and the last node points to nullptr (indicating the end of the list).

1.1.4 Traversing

Traversing a list is simple.

```
node* curr = head;

while (curr) {
    curr = curr->next;
    ...
}
```

1.1.5 Printing

Now that we can traverse, we can print each node

```
node* curr = head;
while (curr) {
    cout << curr->data;
    curr=curr->next;
}
```

1.1.6 Printing in reverse

Printing in reverse requires creating a stack.

```
if (!head) return; // noop, dont even bother creating a vector.

vector<node*> stack;
node* curr = head;

while (curr) {
    stack.push_back(curr);
    curr=curr->next;
}

for (int i=(int)stack.size()-1; i>=0; --i) {
    cout << stack[i]->data << " ";
}
cout << endl;
```

1.1.7 Getting the length

While we traverse, just increment a counter.

```
size_t len() {
    size_t len = 0;
    for (node* curr = head; curr; curr=curr->next, ++len);
    return len;
}
```

1.1.8 Clearing

```
void clear() {
    node* curr=head, *prev=nullptr;

    while (curr) {
        prev=curr;
        curr=curr->next;
        delete prev;
    }
    head = nullptr;
}
```

1.1.9 Reversing

Reversing is pretty straight forward

```
void reverse() {
    node* prev=nullptr, *curr=head, *next=nullptr;

    while(curr) {
        next=curr->next;
        curr->next = prev;
        prev = curr;
        curr=next;
    }

    head = prev;
}
```

In each iteration, next temporarily holds the next node so you don't lose track of it when reversing the link.

The curr->next pointer is set to prev, effectively reversing the link.

Prev is then updated to curr, and curr is updated to next to continue the process.

1.1.10 Pushing

```
void push(int element) {
    if (!head) {
        head = new node(element);
        return;
    }

    node* curr = head;
    while (curr->next) {
        curr=curr->next;
    }
    curr->next = new node(element);
}
```

1.1.11 Inserting

```
void insert(int pos, int element) {
    if (!head || pos == 0) {
        node* new_node = new node(element);
        new_node->next = head;
        head = new_node;
        return;
    }
    node* curr = head;

    int count=0;
    while (count != pos-1 && curr->next) {
        curr=curr->next;
        ++count;
    }
    node* new_node = new node(element);

    new_node->next = curr->next;
    curr->next = new_node;
}
```

1. **Check if the list is empty or inserting at the head (position 0):**
   - If head is nullptr (meaning the list is empty) or pos == 0 (you want to insert at the beginning), a new node is created with the given element.
   - The new node's next pointer is set to the current head (which could be nullptr if the list is empty), and then head is updated to point to this new node.
   - This handles the case where the new node becomes the first node in the list.
2. **Traverse to the correct position:**
   - If you are inserting somewhere other than the head, the function uses a loop to find the node just before the desired position (pos - 1).
   - It starts at the head and moves along the list until it reaches the node right before where the new node will be inserted.
3. **Insert the new node:**
   - Once the loop finds the right place (curr points to the node before the insertion position), a new node is created.
   - The new node's next pointer is set to curr->next (the node currently in the target position).
   - Then, curr->next is updated to point to the new node, effectively inserting the new node into the list.

1.1.12 Popping

```
void pop() {
    if (!head) return;
    if (!head->next) {
        delete head;
        head=nullptr;
        return;
    }

    node* prev=nullptr, *curr = head;
    while (curr->next) {
        prev=curr;
        curr=curr->next;
    }
    delete curr;
    prev->next=nullptr;
}
```

1. **Empty List Check:** If the list is empty (head == nullptr), it does nothing.
2. **Single Node Case:** If the list has only one node, it deletes the head and sets head to nullptr.
3. **Multiple Nodes:** It traverses to the last node using two pointers (prev and curr), deletes the last node (curr), and sets the second-to-last node's next pointer (prev->next) to nullptr to mark the new end of the list.

1.1.13 Erasing

```
void erase(int element) {
    if (!head) return;

    while (head->data == element) {
        if (head->next && head->data == element) {
            node* tmp = head;
            head = head->next;
            delete tmp;
        }
    }

    node* prev=nullptr, *curr=head;

    while (curr) {
        if (curr->data == element) {
            node* tmp = curr;
            prev->next = curr->next;
            curr=curr->next;
            delete tmp;
        } else {
            prev=curr;
            curr=curr->next;
        }
    }
}
```

This erase function removes all nodes with a specific value (element) from the list:

- **Empty List Check:** If the list is empty (head == nullptr), it returns immediately.
- **Head Node Deletion:** If the head contains the target value, it deletes the head and updates it to the next node. We keep doing this until the head node no longer contains the data we want to remove
- **Traverse and Delete:** It iterates through the list, and for each node with the target value, it removes the node by adjusting the next pointer of the previous node and deleting the current node.

1.1.14 Searching

```
node* search(int element) {
    node* curr = head;
    while (curr) {
        if (curr->data == element) {
            return curr;
        }
    }
    return nullptr;
}
```

Recursion

2.1 Elementary recursion

A recursive definition consists of two parts. In the first part, called the anchor or the ground case, the basic elements that are the building blocks of all other elements of the set are listed. In the second part, rules are given that allow for the construction of new objects out of basic elements or objects that have already been constructed. These rules are applied again and again to generate new objects. For example, to construct the set of natural numbers, one basic element, 0, is singled out, and the operation of incrementing by 1 is given as:

1. $0 \in \mathbb{N}$
2. If $n \in \mathbb{N},\ then (n + 1) \in \mathbb{N}$
3. There are no other objects in the set $\mathbb{N}$

It is more convenient to use the following definition, which encompasses the whole range of Arabic numeric heritage:

1. $0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \in \mathbb{N}$
2. If $n \in \mathbb{N}$, then $n0, n1, n2, n3, n4, n5, n6, n7, n8, n9 \in \mathbb{N}$
3. These are the only natural numbers

Recursive definitions serve two purposes: generating new elements, as already indicated, and testing whether an element belongs to a set. In the case of testing, the problem is solved by reducing it to a simpler problem, and if the simpler problem is still too complex it is reduced to an even simpler problem, and so on, until it is reduced to a problem indicated in the anchor

2.2 Base cases

In recursion, a base case is a condition that stops further recursive calls and provides a direct answer without further recursion

If there were no base case, there would be nothing to stop the recursion. Thus, it would go on until the program crashes. For this reason, all recursive functions must have at least one base case.

If a base case in a recursive function returns a value, then every recursive call leading up to that base case should also return a value. This is necessary to ensure that the result of the recursion is propagated back up the call stack.

In a recursive function, the base case stops the recursion, and if the base case returns something (e.g., a node pointer, integer, etc.), the recursive calls that occur before reaching the base case need to return that result so it can propagate back to the original caller.

2.2.1 Factorials

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n \neq 0 \end{cases}.$$

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n-1);

    // Expands to
    // n * n-1 * n-2 * ... * 1
}
```

2.2.2 Powers

Consider the recursive definition for a power of x

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}.$$

```
constexpr int power(int x, int n) {
    if (n == 0) return 1;
    return x * power(x,n-1);
}
```

The function power() can be implemented differently, without using any recursion, as in the following loop:

```
int power2(int x, int n) {
    int res = 1;

    for (res = x; n > 1; --n) {
        res*=x;
    }
    return res;
}
```

Do we gain anything by using recursion instead of a loop? The recursive version seems to be more intuitive because it is similar to the original definition of the power function. The definition is simply expressed in C++ without losing the original structure of the definition. The recursive version increases program readability, improves self-documentation, and simplifies coding. In our example, the code of the nonrecursive version is not substantially larger than in the recursive version, but for most recursive implementations, the code is shorter than it is in the nonrecursive implementations

2.3 Tail recursion

Tail recursion is a type of recursion where the recursive call is the last thing the function does before returning a result. This means there are no more computations or operations to perform after the recursive call.

Because of this, tail recursion can be optimized by some compilers or interpreters to avoid adding new frames to the call stack, making it more memory-efficient than regular recursion.

In simple terms, if a recursive function calls itself, and after that call there's nothing left to do, it's tail recursion. This allows the function to reuse the same memory space, preventing stack overflow in cases with deep recursion.

the recursive call is not only the last statement but there are no earlier recursive calls, direct or indirect. For example, the function tail() defined as

```
void tail(int i) {
    if (i > 0) {
        cout << i << '';
        tail(i-1);
    }
}
```

Is an example of a function with tail recursion, whereas the function nonTail() defined as

```
void nonTail(int i) {
    if (i > 0) {
        nonTail(i-1);
        cout << i << '';
        nonTail(i-1);
    }
}
```

Is not. Tail recursion is simply a glorified loop and can be easily replaced by one. In this example, it is replaced by substituting a loop for the if statement and decrementing the variable i in accordance with the level of recursive call. In this way, tail() can be expressed by an iterative function:

```
void iterativeEquivalentOfTail(int i) {
    for ( ; i > 0; i--)
    cout << i << '';
}
```

Is there any advantage in using tail recursion over iteration? For languages such as C++, there may be no compelling advantage, but in a language such as Prolog, which has no explicit loop construct (loops are simulated by recursion), tail recursion acquires a much greater weight. In languages endowed with a loop or its equivalents, such as an if statement combined with a goto statement, tail recursion should not be used.

Another problem that can be implemented in recursion is printing an input line in reverse order. Here is a simple recursive implementation:

```
void reverse() {
    char ch;
    cin.get(ch);
    if (ch != '\n') {
        reverse();
        cout.put(ch);
    }
}
```

Compare the recursive implementation with a nonrecursive version of the same function:

```
void simpleIterativeReverse() {
    char stack[80];
    int top = 0;
    cin.getline(stack,80);
    for (top = strlen(stack) - 1; top >= 0;
    ↪ cout.put(stack[top--]));
}
```

functions like strlen() and getline() from the standard C++ library can be used. If we are not supplied with such functions, then our iterative function has to be implemented differently:

```
void iterativeReverse() {
    char stack[80];

    register int top = 0;
    cin.get(stack[top]);

    while(stack[top]!='\n') {
        cin.get(stack[++top]);
    }
    for (top -= 2; top >= 0; cout.put(stack[top--]));
}
```

2.4 Indirect Recursion

The preceding sections discussed only direct recursion, where a function $f()$ called itself. However, $f()$ can call itself indirectly via a chain of other calls. For example, $f()$ can call $g()$, and $g()$ can call $f()$. This is the simplest case of indirect recursion. The chain of intermediate calls can be of an arbitrary length, as in:

$$f() \to f_1() \to f_2() \to ... \to f_n() \to f().$$

There is also the situation when $f()$ can call itself indirectly through different chains. Thus, in addition to the chain just given, another chain might also be possible. For instance

$$f() \to g_1() \to g_2() \to ... \to g_m() \to f().$$

This situation can be exemplified by three functions used for decoding information. receive() stores the incoming information in a buffer, decode() converts it into legible form, and store() stores it in a file. receive() fills the buffer and calls decode(), which in turn, after finishing its job, submits the buffer with decoded information to store(). After store() accomplishes its tasks, it calls receive() to intercept more encoded information using the same buffer. Therefore, we have the chain of calls

$$\text{recieve}() \to \text{decode}() \to \text{store}() \to \text{recieve}() \to \text{decode}() \to ....$$

2.5 Nested Recursion

A more complicated case of recursion is found in definitions in which a function is not only defined in terms of itself, but also is used as one of the parameters. The following definition is an example of such a nesting

$$h(n) = \begin{cases} 0 & \text{if } n = 0 \\ n & \text{if } n > 4 \\ h(2 + h(n)) & \text{if } n \leqslant 4 \end{cases}.$$

2.6 Excessive Recursion

Logical simplicity and readability are used as an argument supporting the use of recursion. The price for using recursion is slowing down execution time and storing on the run-time stack more things than required in a nonrecursive approach. If recursion is too deep (for example, computing $5.6^{100,000}$), then we can run out of space on the stack and our program crashes. But usually, the number of recursive calls is much smaller than 100,000, so the danger of overflowing the stack may not be imminent

However, if some recursive function repeats the computations for some parameters, the run time can be prohibitively long even for very simple cases

Consider Fibonacci numbers. A sequence of Fibonacci numbers is defined as follows:

$$\text{Fib}(n) = \begin{cases} n & \text{if } n < 2 \\ \text{Fib}(n-2) + \text{Fib}(n-1) & \text{otherwise} \end{cases}.$$

The definition states that if the first two numbers are 0 and 1, then any number in the sequence is the sum of its two predecessors. But these predecessors are in turn sums of their predecessors, and so on, to the beginning of the sequence.

How can this definition be implemented in C++? It takes almost term-by-term translation to have a recursive version, which is

```
constexpr unsigned long fib(int n) {
    if (n < 2) return n;
    return fib(n-2) + fib(n-1);
}
```

The function is simple and easy to understand but extremely inefficient. To see it, compute Fib(6), the seventh number of the sequence, which is 8. Based on the definition, the computation runs as follows:

$$\begin{aligned} Fib(6) &= Fib(4) + Fib(5) \\ &= Fib(2) + Fib(3) + Fib(5) \\ &= Fib(0) + Fib(1) + Fib(3) + Fib(5) \\ &= 0 + 1 + Fib(3) + Fib(5) \\ &= 1 + Fib(1) + Fib(2) + Fib(5) \\ &= 1 + Fib(1) + Fib(0) + Fib(1) + Fib(5). \end{aligned}$$

Etc... The source of this inefficiency is the repetition of the same calculations because the system forgets what has already been calculated. For example, Fib() is called eight times with parameter n = 1 to decide that 1 can be returned. For each number of the sequence, the function computes all its predecessors without taking into account that it suffices to do this only once.

It takes almost a quarter of a million calls to find the twenty-sixth Fibonacci number, and nearly 3 million calls to determine the thirty-first! This is too heavy a price for the simplicity of the recursive algorithm. As the number of calls and the run time grow exponentially with n, the algorithm has to be abandoned except for very small numbers

An iterative algorithm may be produced rather easily as follows:

```
unsigned long iterativeFib(unsigned long n) {
    if (n < 2)
    return n;
    else {
        register long i = 2, tmp, current = 1, last = 0;
        for ( ; i <= n; ++i) {
            tmp = current;
            current += last;
            last = tmp;
        }
        return current;
    }
}
```

However, there is another, numerical method for computing Fib(n), using a formula discovered by Abraham de Moivre:

$$\mathrm{Fib}(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}.$$

Where $\phi = \frac{1}{2}(1 + \sqrt{5})$, and $\hat{\phi} = 1 - \phi = \frac{1}{2}(1 - \sqrt{5})$. $\hat{\phi}$ becomes very small when n grows, thus it can be omitted.

$$\mathrm{Fib}(n) = \frac{\phi^n}{\sqrt{5}}.$$

Approximated to the nearest integer

```
unsigned long deMoivreFib(unsigned long n) {
    return ceil(exp(n*log(1.6180339897) - log(2.2360679775)) -
    ↪  .5);
}
```

2.7 Backtracking

In solving some problems, a situation arises where there are different ways leading from a given position, none of them known to lead to a solution. After trying one path unsuccessfully, we return to this crossroads and try to find a solution using another path. However, we must ensure that such a return is possible and that all paths can be tried. This technique is called backtracking, and it allows us to systematically try all available avenues from a certain point after some of them lead to nowhere. Using backtracking, we can always return to a position that offers other possibilities for successfully solving the problem. This technique is used in artificial intelligence, and one of the problems in which backtracking is very useful is the eight queens problem.

The eight queens problem attempts to place eight queens on a chessboard in such a way that no queen is attacking any other To solve this problem, we try to put the first queen on the board, then the second so that it cannot take the first, then the third so that it is not in conflict with the two already placed, and so on, until all of the queens are placed. What happens if, for instance, the sixth queen cannot be placed in a nonconflicting position? We choose another position for the fifth queen and try again with the sixth. If this does not work, the fifth queen is moved again. If all the possible positions for the fifth queen have been tried, the fourth queen is moved and then the process restarts. This process requires a great deal of effort, most of which is spent backtracking to the first crossroads offering some untried avenues. In terms of code, however, the process is rather simple due to the power of recursion, which is a natural implementation of backtracking

```
putQueen(row)
    for every position col on the same row
        if position col is available
            place the next queen in position col;
            if (row < 8)
                putQueen(row+1);
            else success;
            remove the queen from position col;
```

This algorithm finds all possible solutions without regard to the fact that some of them are symmetrical.

2.8 Recursion in singly linked lists

2.8.1 Traversing

To traverse a linked list using recursion, you need to define a recursive function that processes the current node and then calls itself with the next node until the list is fully traversed (i.e., until the current node is nullptr).

```
void TraverseList(node* head) {
    if (!head) {
        return;
    }
    TraverseList(head->next);

    // ...
}
```

2.8.2 Printing

We can use this, for example, to print each nodes data member

```
void PrintList(node* head) {
    if (!head) return;

    cout << head->data << " ";
    PrintList(head->next);
}
```

2.8.3 Printing in reverse

We a slight alter in the print example, we can reverse print the list.

```
void PrintListReverse(node* head) {
    if (!head) return;

    PrintListReverse(head->next);
    cout << head->data << " ";
}
```

2.8.4 Getting the length

2.8.5 Clearing

We can also use this to clear the list

```
void clear() {
    std::function<void(node*)> r_clear = [&] (node* p) = {
        if (!head) return;

        r_clear(head->next);
        delete head;
    }
    r_clear(head);
    head=nullptr;
    size=0;
}
```

2.8.6 Reversing

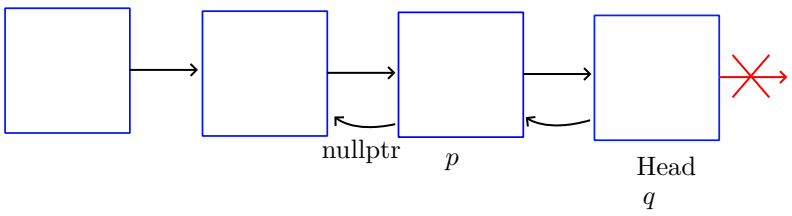
Let's first take a look at the reverse code

```
1  void reverse() {
2      std::function<void(node*)> r_reverse = [&] (node* p) -> void
   ↪  {
3          if (!p->next) {
4              head = p;
5              return;
6          }
7
8          r_reverse(p->next);
9          node* q = p->next;
10         q->next = p;
11         p->next = nullptr;
12
13     };
14     r_reverse(head);
15 }
```

The base case is that we are at the end, in this case we set head to this position. Head is now at the end of the list.

Once the base case is triggered and the head is set to the last node in the list, we will be sent back to the n-1 node call.

To get the intuition for linked list logic, we must examine a diagram of the list.



This figure shows the three operations done after each recursive call. In the figure above, we are at the node after the call that set the end to head. We

1. Get a pointer to the node ahead of the current (q).
2. This allows us to severe its old next pointer and reverse its direction.
3. Then, set p next to nullptr (set up for next return).

When the callstack returns to the first call, and does its operations, the list will be reversed

It is also a good idea to examine the iterative method.

```
void Itreverse() {
    node* prev=nullptr, *curr=head, *next=nullptr;

    while (curr) {
        next=curr->next; // Move next to the next node
        curr->next=prev; // Change the direction of current
↪   nodes next pointer

        prev=curr; // Advance prev
        curr=next; // Advance curr
    }
    head=prev; // Prev is last node, set head to end
}
```

2.8.7 Pushing

```
void push(int data) {
    if (!head) {
        head = new node(nullptr, data);
        return;
    }
    std::function<void(node*, int)> r_push = [&] (node* curr,
    ↪  int data) -> void {

        if (!curr->next) {
            curr->next = new node(nullptr, data);
            ++size;
            return;
        }
        r_push(curr->next, data);
    };
    r_push(head, data);
}
```

Base case:

1. **Empty list**: No recursion, make head the new node

Otherwise, recurse from the head until we get to the last node, simply set last nodes next pointer to new node and return

2.8.8 Inserting

```
void insert(unsigned pos, int element) {
    std::function<void(node*&, unsigned)> r_insert = [&] (node*&
    ↪  p, unsigned curr_pos) {
        if (curr_pos == 0) {
            node* new_node = new node(nullptr, element);
            new_node->next = p;
            p = new_node;
            return;
        }
        r_insert(p->next,  curr_pos-1);
    };
    r_insert(head, pos);
}
```

Base case

1. **Recursed the same number of times as the pos arg**: In this case, make a new node, set next to current node in the recursive traversal, set current node to new node.

Otherwise, keep recursing, subtracting one from the curr_pos.

2.8.9 Popping

```
void pop() {
    if (!head) return;

    if (!head->next) {
        delete head;
        head=nullptr;
        return;
    }

    std::function<void(node*)> r_pop = [&] (node* p) -> void {
        if (!p->next->next) {
            delete p->next;
            p->next = nullptr;
            --size;
            return;
        }
        r_pop(p->next);
    };
    r_pop(head);
}
```

Base cases:

1. **Empty list**: Noop
2. **One node (head)**: Delete then reset head

Otherwise, recurse until we are at the second to last node. Then, delete the second to last nodes next node, which is the last node. Set second to last nodes next pointer to nullptr.

2.8.10 Erasing

```
void erase(int element) {
    std::function<void(node*&)> r_erase = [&] (node*& p) -> void
    {
        if (p == nullptr) {
            return;
        }

        r_erase(p->next);

        if (p->data == element) {
            node* tmp = p;
            p = p->next;
            delete tmp;
        }
    };
    r_erase(head);
}
```

Base case:

1. **Reached the end**: Return, start unwinding

We traverse to the end of the list recursively, once we reach the end the recursion stops and we start unwinding the call stack, going backwards in the list.

For each node, we check if its data is equal to the element, if it is we set this node equal to its next node, then delete.

2.8.11 Searching

```
node* search(int element) {
    std::function<node*(node*)> r_search = [&] (node* p) ->
    node* {
        if (p == nullptr)  {
            return nullptr;
        }
        if (p->data == element) {
            return p;
        }
        return r_search(p->next);
    };
    return r_search(head);
}
```

Base cases:

1. **Reached the end of the list**: Element is not in list, return nullptr
2. **Found the first node with the element**: Return the node

Otherwise, recurse through the nodes until we hit one of the base cases.