Linking with Linked Lists

Greg Zborovsky

Abstract

Data structure courses often introduce linked lists as a basic data structure as an alternative to arrays. They supposedly have the advantage of constant time insertion and deletion at any point in the list as long as you have access to a pointer near the node you are inserting or deleting. However, they suffer from cache and implementation issues that array-backed data structures tend to avoid.

In this paper, I will

- Reintroduce linked lists and their implementations
- Reintroduce array-backed data structures and their implementations
- Discuss issues present with linked lists
- Discuss potential and practical applications of linked lists

If there is anything you should take away from this paper, it is the content in section 4, "Why it's Worse Than You Think"

1 Introduction

Because this paper will focus primarily on speed differences between array-backed data structures and linked list-backed data structures, the majority of the material will be written with the intent of using a lower-level programming language like C++, although I will also mention Java performance.

By far, the most popular array-backed data structure in C++ is std::vector, the closest alternative to Java's ArrayList. It is important to keep in mind that arrays are of a fixed size, so if you end up inserting an element when no slots are left, an $O(n)$ operation must occur to relocate all of the elements to a larger array. Despite this "worst case" $O(n)$ behavior, we can add to the end of a vector in $O(1)$ amortized time, since it will still take $O(n)$ time to insert n elements at the end. Note that in this implementation, it will always take $O(n)$ time to insert an element at an arbitrary position, such as the beginning of the vector.

C++'s equivalent linked list implementation is std::list, which is the closest alternative to Java's LinkedList. Linked lists (assuming any kind, such as singly, doubly, circularly, etc) have the advantage of a true $O(1)$ time insert operation near any node which we have a pointer to. In the default implementation, this means we can do a true $O(1)$ insert to either side. Despite the true $O(1)$ time

insert operation, I will explore why linked lists often offer slower performance than array-backed data structures.

2 Array-Backed Data Structures

For the purpose of this paper, I would like to primarily use a C++ implementation of Java's ArrayDeque for argument about the efficiency of array-backed data structures. C++ has std::deque in the standard library, but uses a slightly more complex implementation which is unnecessary for this paper. The common operations of an ArrayDeque will have the time complexities listed below:

Operation	Time
Insert at beginning	$O(1)$ amortized
Remove from beginning	$O(1)$
Insert at end	$O(1)$ amortized
Remove from end	$O(1)$
Insert at arbitrary index	$O(n)$
Remove from arbitrary index	$O(n)$
Access at arbitrary index	$O(1)$
Search for element	$O(n)$

Note that operations related to access, insertion, and deletion on either side are all $O(1)$ or amortized $O(1)$, making our ArrayDeque implementation efficient for the purposes of a deque (double ended queue). [A double ended queue may be used as both a stack and a queue.]

Our ArrayDeque is implemented by maintaining an array of a certain size m, the start index of elements, and the end index of elements. On inserting at the end of the data structure, we will place an element at the end index and increment end index by 1 and mod it by m. For this reason, it may be preferable to constrain m to be a power of two such that the expensive mod operation can be replaced with a cheap bitmask. If end index is equal to start index, we will double the size of the containing array and copy over all elements. Similarly, when we place an element at the beginning, we will subtract 1 from the index and mod it by m. If the new start index is equal to the end index, we will double the size of the containing array and copy over all elements. We will then place the element at the new start index. Similar to the usual analysis of std::vector and ArrayList, adding to either side will be $O(1)$ amortized time because n operations will take $O(n)$ time. [If we assume that we double the size of the backing array, then over n operations, there will be n inserted elements and roughly $\log_2(n)$ array resizes. If the final array size is s, then the total number of allocated array slots over all n operations would be at most $2s$, which is still linear. This same analysis still applies if array size is multiplied by a value other than two as long as it is strictly greater than one, but note that the constants used in this analysis will be off. Big-O asymptotic complexities would remain the same.]

To remove elements on either side, we can simply increment or decrement the start and end indices, both of which clearly take $O(1)$ time.

We can maintain $O(1)$ time random access by index by returning the value at $(start + i)\%m$ where i is the index we want to access, m is the current size of the array, and *start* is the start index. Asymptotically, this is equivalent to the time complexity for the same operation for an ArrayList or for an std::vector. If m is a power of 2, then performance in practice is very similar as well. Note that this operation is impossible to implement for linked list-backed data structures at this time complexity.

3 Linked List-Backed Data Structures

There are several different implementations of linked lists, such as singly linked lists, doubly linked lists, circularly linked lists, XOR linked lists, and in-array linked lists, but I will primarily focus on doubly linked lists. C++ has its own std::list, but I will implement my own doubly linked list for simplicity.

I will assume the reader knows how to implement a double linked list with a next and a prev pointer, but will specify the struct for a node:

```
1 template <class T>
2 struct node {
3     node* prev;
4     node* next;
5     T data;
6 };
```

Note that `T data` is a value and not a pointer so as to not introduce another level of indirection. Interestingly enough, making the data member a value instead of a pointer (which isn't nearly as easy in some languages, such as Java) makes the benefits of the linked list much more like those of intrusive linked lists (although not the same). The common operations of a doubly linked list will have the time complexities listed below:

Operation	Time
Insert at beginning	$O(1)$
Remove from beginning	$O(1)$
Insert at end	$O(1)$
Remove from end	$O(1)$
Insert at arbitrary index	$O(n)$
Insert a node "near" another	$O(1)$
Remove from arbitrary index	$O(n)$
Remove an arbitrary node	$O(1)$
Access at arbitrary index	$O(n)$
Search for element	$O(n)$

4 Why it's Worse Than You Think

4.1 In Theory

Note that although a doubly linked list can remove a given node in $O(1)$ time or insert a new value after or before a given node in $O(1)$, finding the node to insert or delete from may take $O(n)$ time in the first place. All other time complexities of linked list operations are either equal to or worse than those of the ArrayDeque (assuming that amortized $O(1)$ is counted as equivalent to true $O(1)$). Even people like Bjarne Stroustrup have noted and presented on this supposed issue (https://www.youtube.com/watch?v=YQs6IC-vgmo).

Given that ArrayDeques have the advantage of $O(1)$ access of arbitrary indices and appear conceptually simpler, it's hard to find a reason to resort to using linked lists. Furthermore, ArrayDeques are backed by arrays, which means that all of the values of an ArrayDeque are all layed out compact and contiguously in memory. When adding to ArrayDeques, and especially when iterating through them, the CPU will find it much easier to load entire chunks of as much of the underlying array into cache at a time.

On the other hand, the nodes of a linked list aren't guaranteed to be anywhere near each other in memory and will require extra overhead to maintain their `prev` and `next` pointers. The extra 16 bytes of memory usage (on an architecture with 8 byte addresses) and potentially random locations in memory will lead to more frequent cache misses, evictions, and less data will fit into the cache per load because the number of bytes used to represent the same amount of information increases, but cache size does not increase. Especially consider that for a containter with very small items, such as 32 bit integers, vectors will have a near-0 overhead for a large number of items, but linked lists will require around 200% more space. Since reads and writes to main memory are many times slower than to cache, programs should optimally try to be as memory and cache efficient as possible.

Linked lists have yet another downside—frequent memory allocations. Typical linked list-backed data structures will allocate memory for each node, likely internally with something similar to a call to `malloc`. When calling `malloc`, the internal library may result to either a slow system call or to pulling a chunk of memory currently managed by `malloc`. Although the latter option is faster and therefore preferred, early allocations in a program will likely use the former option. Even when `malloc` takes the faster, latter path, in typical default implementations, it must still hop through a many lines of code, and if using multiple threads, a mutex as well.

Although more efficient memory allocators exist, such as `tcmalloc` and `ptmalloc2`, the benefits of reducing the total calls to memory allocators is clear. Where adding $O(n)$ elements to a linked list requires roughly $O(n)$ memory allocations, adding $O(n)$ elements to an ArrayDeque requires roughly $O(\log n)$ memory allocations (because memory allocations for an ArrayDeque are only done when its capacity is doubled).

4.2 In Practice

4.2.1 Context

The regular tests are run on collections of 4 byte integers, whereas the "large" version of the same test is run with a class whose only member is std::array<int, 128>, which means that the entire class will be roughly 512 bytes. It is likely important to note that these timing tests are single-threaded tests run on an i5-1135G7 running at 2.40 GHz on Windows through WSL and compiled with -O3 in g++.

4.2.2 Results

	Array Deque (s)	Linked List (s)	Difference
push_back	0.102	0.763	7.5x
push_front	0.141	0.736	5.2x
iterate	0.011	0.814	74x
push_back large	0.15	0.143	0.95x
push_front large	0.132	0.082	0.62x
iterate large	0.012	0.06	5.0x

4.2.3 Caution

Each timing is given in seconds, and the rightmost column is just the linked list column divided by the array deque column. The values in the table above are only rough numbers, but give a good sense of the expected performance given certain input data. For a small element size, each test was conducted 100 times with roughly 33 million elements inserted. For a large element size, each test was conducted 30 times with roughly 250,000 elements inserted.

Note that for a small element size, the performance of the array deque totally crushes that of the linked list. For push_back and push_front, which is $O(1)$ amortized for array deques and $O(1)$ for linked lists, the array deque tends to be around 5 times faster. When it comes to iteration, the vector is even better suited, and ends up more than 50 times faster than a linked list.

For a larger element size, the overhead in the inherent structure of a linked list is small relative to the size of the element. Additionally, linked lists only need to store exactly as many nodes as elements have been added and do not need additional capacity. Thus, for a larger element size, a linked list of values *may* take less space than an array deque of values (Although this may be somewhat remedied by changing the array deque to store pointers to values rather than the values themselves). At large sizes, linked lists may use slightly less memory and for adding elements, may even be slightly faster. However, for iteration, linked lists still lag far behind array-backed data structures.

While specific timings should not be used and may depend on other running processes, OS, CPU, compiler, and so on, the general trends are to be trusted.

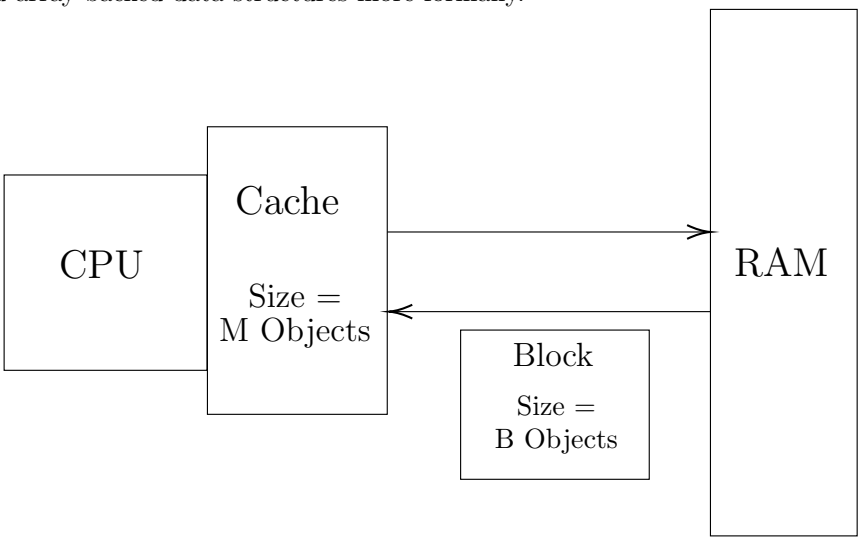
4.3 Additional Analysis

A common mistake is to assume that since iterating over n elements takes $O(n)$ time for both a linked list and an array-backed data structure that their actual execution times in practice should also be similar. In practical tests, we find a massive disparity. In order to begin addressing this, we introduce the external-memory model. This model is often used to analyze how often IO operations are done to load chunks of data from a hard disk to CPU cache or RAM, but we can similarly use it to analyze loads of block from memory (RAM) to a much smaller cache.

We assume that RAM is infinitely sized, but that the cache may contain M objects and blocks which contain B objects can be loaded. We assume that $B < M$ by a large margin. N is the number of objects.

For iteration over an array, the number of cache load operations is $O(\frac{N}{B})$ since contiguous elements can be loaded in a single load to cache. However, for a linked list, the number of cache load operations is $O(N)$ since none of the nodes are guaranteed to be close to each other in memory, so each node access only necessarily loads itself into cache from RAM. Given that cache loads are a relatively expensive calculations, the division by B is an impactful performance improvement.

This still doesn't illuminate the entire picture, but provides an initial avenue through which we can analyze the performance differences between linked lists and array-backed data structures more formally.



5 Why it's Better Than You Think

Even despite all the listed downsides of linked lists, it can still importantly perform an operation that arrays cannot. Linked lists can insert or remove a node "near" a given node in $O(1)$, so if we can somehow access a given node in

faster than $O(n)$ time, then we may end up with a situation where any array-backed data structure must be slower.

One idea to accelerate access to any node is to add a lookup table, like a HashTable, where any inserted node is also added to the lookup table with an appropriate key. The restriction in this case is that each node must have a logical key associated with it that isn't an index (since we can't update the index efficiently once we place it into the lookup table). Of course, the worst-case lookup time for a HashTable is technically $O(n)$, but for any decent hash function, we can expect $O(1)$ time behavior on average. If stronger guaranteed bounds are absolutely necessary, we can resort to a tree-backed lookup table with $O(log(n))$ lookup and insert times, which still beats $O(n)$ time lookups. The advantage of augmenting a linked list with a lookup table is being able to access and modify any portion of your data *and* still maintain an order to it.

Even despite the cache and memory benefits of array-backed containers, the $O(1)$ performance with linked lists of the above use case very quickly beats that of the $O(n)$ performance of arrays-backed data structures.

There exists another previously unmentioned benefit of linked lists regarding memory as well. For a small data type like 32 bit integers, linked lists may use 200% more memory than vectors, but for a large datatype, linked lists will beat array-backed data structures in memory efficiency. Although linked lists require a minimum overhead of 16 bytes per element, they only allocate exactly as much memory as they need to store the list, but an array-backed container whose size occasionally doubles may need to allocate up to 2x more memory than it actually uses at a given time. When the size of each item is large, the memory overhead of array-backed containers dominates that of linked lists.

6 Applications

6.1 Generalized Use Case

As far as I understand, these conditions must hold for a linked list or linked list-like data structure to be required or very helpful to achieve optimal asymptotic time complexity:

1. You need to insert, move, or delete nodes in the middle of a list.
2. You require ordered (not sorted) data.
3. The elements in your data must have some innate property such that you can access them in faster than $O(n)$ time where n is the number of elements in the data.
4. The problem must never require the data to be indexed.

6.2 LRU Cache

Frequently seen as an interview question, using the combination of a HashTable and a LinkedList is the most simple and efficient way to solve this problem.

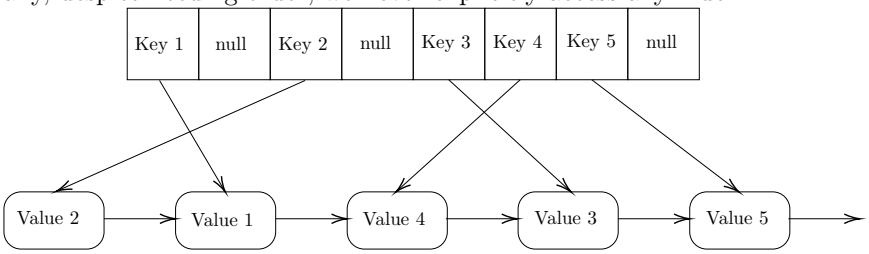
Design a data structure with a capacity c such that you can insert key-value pairs into this data structure and then query for the value of any given key. If the user inserts an item when the number of items in the data structure has already reached c, then evict the element that was accessed the longest time ago. Ensure that all operations run in roughly $O(1)$ time.

With just a hash table, inserting when capacity is below c and querying will both run in roughly $O(1)$ time, but the process of removing an element once the number of elements exceeds c will take roughly $O(n)$ time to find which element hasn't been accessed in the longest time. With some tree-like data structure, this may be sped up to $O(log(n))$, but not $O(1)$.

With just an array, inserting can be fully implemented in $O(1)$, but querying will be stuck at $O(n)$.

We can combine the best traits of both worlds by augmenting a linked list with a lookup table as described in section 5. Every time we need to insert a key-value pair, we simply add the key-value pair to the end of the linked list and add a new entry to the table where the key is the given key and the value is a pointer to the newly added linked list node. If after this procedure the size of our data structure exceeds c, we can simply remove the first element of the linked list and remove its associated key from the lookup table. To query, if the given key is not in the lookup table, then we do nothing. If it is indeed in the lookup table, then with the pointer to the node, we remove it in $O(1)$ time and re-add it to the end of the list. This new structure supports both insertion and querying in roughly $O(1)$ time and relies on both key-based lookups and having a particular order to the data.

Note that all 4 conditions mentioned in section 6.1 are satisfied here. For the query operation, we delete nodes from the middle of a list. To know which element was accessed the longest ago, we need to maintain order. The innate property by which we can access each node is the key from each key-value pair. Finally, despite needing order, we never explicitly access any index.



6.3 malloc

Most implementations of malloc, such as linux's and the freeBSD's, will use some form of linked lists to store which chunks of memory are free. As noted in section 4, iteration over linked lists is extremely slow in practice, so all of these implementations refrain from performing any iteration over linked lists.

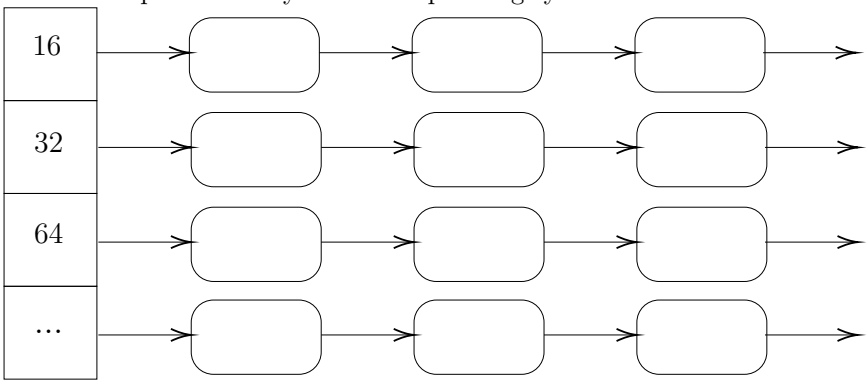
The general idea is that each malloc implementation will keep an array of size n, each with a pointer to the first node of a linked list. Each array index

represents a different size to allocate. For example, the first index in the array may represent free memory blocks of size 16, the next index of 32, then of 64, etc. Thus, the allocator will be able to offer greater granularity in memory allocation than otherwise efficiently possible.

When a caller wants to allocate memory of size s, memory of size 2^{k+4} will be allocated where k is the smallest integer such that $2^{k+4} \geq s$. To allocate this memory, we will use k to index into the array of linked list pointers, we will return a pointer to the memory in only the first node of the linked list, and we will change the pointer at index k of the array to point to the next node. Since the array maintains a pointer to the first node and every node maintains a pointer to the next node, there is no significant iteration.

Interestingly enough, when k is large enough, the memory allocated to store a node itself can be given directly to the caller requesting memory, so after a block of memory is allocated, there is nearly essentially no memory overhead for that block in the allocator.

All of the above already assumes that malloc has already called something like `sbrk` to request memory from the operating system.



6.4 Dancing Links

Dancing links (DLX) is a technique used to very simply insert or remove a node in $O(1)$ into or from a circular doubly linked list. If there already exists a circular doubly linked list, we can remove a node from the list but still keep the node in memory with its previous and next pointers preserved in $O(1)$ time. Since we preserve its previous and next pointers, it can be re-inserted into the same position in the list in $O(1)$ time as well given that the nodes referred to by previous and next pointers are already in the list.

Donald Knuth used this trick in designing Algorithm X, a recursive, backtracking algorithm used to solve exact cover problems. The exact cover problem is to select a subset of the rows of a binary matrix such that a '1' appears exactly once in each column. As an example, a game like Sudoku can be converted to an equivalent exact cover problem.

This paper won't cover how Algorithm X works, but an efficient implementation of it that makes use of DLX will store only the 1s of the binary matrix

in a 2D linked list, so it will be efficient even for sparse matrices. In each step of the algorithm, certain rows and columns of the matrix can be removed very efficiently because of DLX. Similarly, when backtracking, removed rows and columns must be re-inserted into their previous position, which is again very efficient because of DLX.

https://en.wikipedia.org/wiki/Dancing_Links

6.5 Competitive Programming

6.5.1 LRU Cache

We have already previously analyzed the design of an efficient software-based LRU Cache, but it is also worth mentioning that designing an LRU Cache is a common interview and leetcode question.

https://leetcode.com/problems/lru-cache/

6.5.2 Advent of Code

Advent of Code is a set of 25 programming problems released each year from December 1st to 25th. Each problem tends to rely on some concepts from computer science and programming. Part 2 of problem 23 from 2020 is one such problem and happened to require some form of a linked list. Similar to the LRU cache problem, the final answer to the problem relies on the order of the collection, but intermediate steps require efficient access to insertion and deletion at arbitrary values (not indices). Again, by combining a lookup table with a linked list, an efficient solution may be written.

https://adventofcode.com/2020/day/23

6.5.3 Google CodeJam

One of the clever solutions to the problem "Square Dance" from Round 1A in Google Code Jam 2020 also conceptually relies on linked lists. There exists a grid of size $R \times C$ where R is the number of rows and C is the number of columns. On each grid cell is one competitor with a certain score assigned to them. In every round until there are no more eliminations, compare the score of each competitor with the average of the scores of the closest 4 competitors to its left, right, top and bottom directions. For any competitor whose score is lower than the average around them, they are not included in the next round (they are eliminated). Naively, this solution will take roughly $O(R^2C^2(R+C))$ because there may be up to $O(RC)$ rounds, up to $O(RC)$ competitors per round, and up to $O(R+C)$ work to find the average value of the 4 competitors to the left, right, top and bottom of a given competitor.

The final solution to the question ends up at $O(RC)$, but our primary interest lies in how to drop the factor of $O(R+C)$ down to $O(1)$. Rather than storing the grid as an array of arrays, we can choose store it as a 2D linked list where each node is a competitor and maintains pointers to the competitors directly above, below, to the left and the right of it. When eliminating competitors, lets

say competitor a, the down pointer of the node above a should put at the node below a and the up pointer of the node below a should put at the node above a. A similar procedure occurs for the nodes to the left and right of a. By following this procedure for each round, every node can find the average of the 4 nodes surrounding it in $O(1)$ rather than searching out in the 4 directions to find the first non-eliminated competitor in $O(R + C)$.

Although the solution above is simple and relatively intuitive, it can also be simulated with just a regular 2D array where each grid cell would additionally need to store the distance to the next closest left, right, top, and bottom neighbors and never decrement any value. This approach should be $O(1)$ amortized since every cell will perform at most $O(R + C)$ updates to its left, right, top, and bottom neighbors over the entire course of $O(RC)$ rounds. This alternative solution may require some additional analysis.

https://codingcompetitions.withgoogle.com/codejam/round/000000000019fd74/00000000002b1355#problem

7 More Resources and Extensions

7.1 Variations

7.1.1 In-Array Linked Lists

Rather than directly allocating memory for each node, one alternative is store a vector or array-backed data structure of nodes. This offers a few potential advantages.

When this type of list is first generated, assuming that each node is linked to the nodes in the next and previous indices, there is better cache performance since elements are guaranteed to be contiguous, although this only applies before nodes are moved around, added, and deleted. Additionally, since the nodes are in an array and not directly within memory, the `prev` and `next` pointers of each node can be replaced with integers representing the index of the previous and next node. For a list with few enough elements, an integer type smaller than 8 bytes may be selected, so this variation may also be slightly more memory efficient. Given how vectors and array deques tend to grow their memory, this also means that memory allocations will happen much more rarely than in a regular linked list.

However, this method does have notable downsides. Given the index of a node, it is still easy to delete it in $O(1)$ time and it is easy to insert a node "after" or "before" any other node in $O(1)$. But, when deleting a node in the middle of an array, we cannot easily fill the hole that it leaves. Thus, the memory complexity of this variation is not $O(n)$ where n is the number of elements in the list, but rather $O(m)$ where m is the total number of calls to insert.

7.1.2 XOR-Linked Lists

Another novel idea is to store only one pointer rather than two for a doubly linked list. This single pointer would represent the bitwise XOR ($\oplus$) of the `prev` pointer and the `next` pointer. The pointer held by the head of the linked list will just be the `next` pointer because $0 \oplus a = a$. Similarly, the pointer held by the tail of the linked list will just be the `prev` pointer because $a \oplus 0 = a$.

The algorithm to iterate from head to tail may look like the following:

Require: head is a pointer to the head node of the linked list
Require: each node contains two properties: an XOR'd pointer and a value
  currNode $\leftarrow$ head
  prevPointer $\leftarrow$ 0
  while currNode $\neq$ null **do**
    process currNode.value
    nextPointer $\leftarrow$ prevPointer $\oplus$ currNode.pointer
    prevPointer $\leftarrow$ address of currNode
    currNode $\leftarrow$ node at nextPointer
  end while

This approach requires only storing one pointer per node instead of two for a doubly linked list, but requires more logical complexity, is less well known, and is more difficult to augment with a faster lookup method.

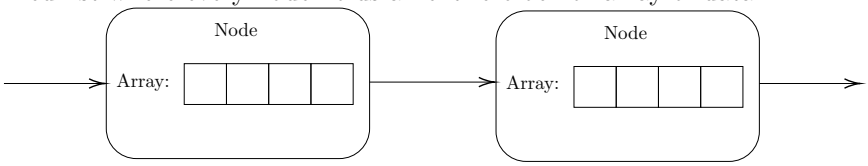
https://en.wikipedia.org/wiki/XOR_linked_list

7.1.3 Unrolled Linked List

One of the issues with the classic, naive linked list is the lack of efficiency with respect to cache, so one of the proposed solutions is to store an array of elements within each node instead of just a single element within each node.

This approach is certainly more cache efficient for access of elements next to each other, but also increases logical complexity and makes adding/deleting nodes in the middle of a list slightly less efficient.

This variation is actually quite common, just hidden as an implementation detail. For example, CPython's built-in deque implementation, as of 3.8.1, uses a linked list where every node holds an entire block or array of data.



https://en.wikipedia.org/wiki/Unrolled_linked_list

7.2 Graphs

Linked lists may be considered trivial graphs. Singly linked lists are directed graphs and doubly linked lists are undirected graphs. If a linked list is circular, then the graph will have a loop. Otherwise, it will be a tree. Since linked lists

are essentially simple graphs, we can represent them with adjacency matrices and adjacency lists. Since they are very sparse graph, adjacency matrices will be very space inefficient, but adjacency lists will be nearly equivalent to the in-array variation of linked lists.

Linked lists typically refer to 1D lists, although they can be extended to 2D just by adding `up` and `down` pointers as well. It is interesting to note that 2D linked list which is circular both vertically and horizontally is actually a torus. We can even continue increasing the number of dimensions, but in 3D and higher, it is simplest to visualize the "linked list" as a graph where we can cheaply insert or delete a node from any position in the graph. (Mention that one Google CodeJam problem)

Normal binary and n-ary trees which have pointers to their children may be thought of as linked lists which branch off as they approach their tails. On the other hand, trees which maintain pointers to their parents may be thought of as linked lists which merge together as they approach their tails. In either case, any simple path in a tree or a graph may also be thought of as a linked list.

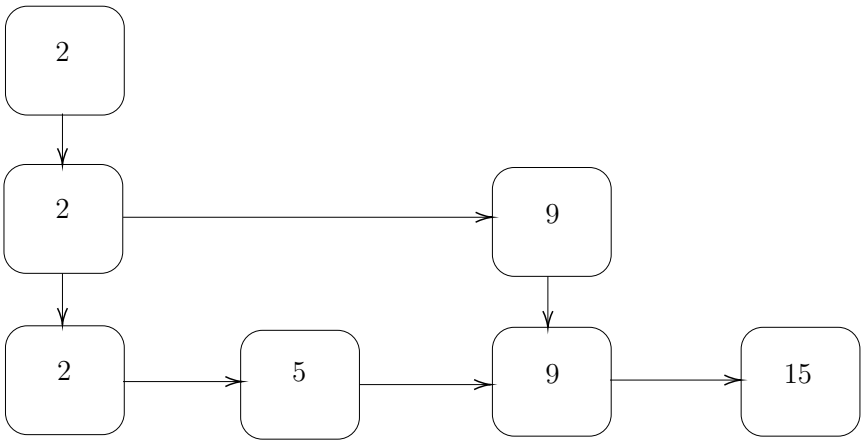
7.3 Skip Lists

For efficient search, insertion, and deletion balanced binary search trees like AVL and red-black trees are often the goto since they implement search, insertion, and delete each in $O(log(n))$ time by maintaining a tree whose inorder traversal is sorted with a depth of $O(log(n))$ where n is the number of elements in the collection.

Skip lists are a data structure based on linked lists where the asymptotic time complexities for search, insertion, and deletion are all $O(log(n))$ as well. However, their time and memory constants are typically higher than those of balanced binary search trees, so they are often ignored.

The general idea for a skip list is to maintain $O(log(n))$ layers of sorted linked lists. Without loss of generality, assume $n = 2^k$. The first layer of the skip list may only contain 1 node, the second layer may contain 2 nodes, the third layer 4 nodes, the the fourth layer 8 nodes, and so on, until it reaches n nodes.

To give a basic idea of why this structure works, we can consider how the search method may work. To search for a value, we will start at the left-most node of the top layer and we will recursively step at most one node to right and exactly one layer down. We only step a node to the right iff the value of the node to the right is less than or equal to the value we are searching for. Thus, we will only take $O(log(n))$ steps to find a value.



https://en.wikipedia.org/wiki/Skip_list

7.4 Closing Thoughts

Despite the better performance of array-backed data structures, there is still something to be said about clean code and simplicity. Even with the existence of array deques, double ended queues tend to be visualized as linked lists. In functional programming, especially when manipulating just the head or tail of a list, it is easier to visualize something akin to a linked list rather than a "view" into an array if we care about achieving optimal or nearly optimal time complexity.

On the note of simplicity, many geometric data structures have very natural representations in a linked list format. For example, the half-edge data structure, also known as a doubly connected edge list, stores information about the face this half of the edge is facing, the next half edge, the previous half edge, and the vertex at the end of edge. The structure of a half edge pointing to the next and previous half edges easily lends itself to a circular doubly linked list-like structure, but most efficient implementations of this data structure tend to avoid directly using linked lists and stick to a design where previous and next half edges are referred to by index. This design is more similar to the in-array variation of linked lists described in section 7.1.1.

An interesting edge case of when array-backed data structures may be unfavorable compared to a linked list-backed data structure is when elements need to be added to the front or back of the container and worst case scenario per insertion is much more important than the average or amortized cost. In a system where real-time and immediate performance must be guaranteed, a linked list may actually be preferable, although I am not familiar with any such scenarios yet.

Another case of where linked list-like structures may shine is in multi-threaded programs when adding or removing elements from a list. With a vector or array-backed data structure, the *entire* data structure must be locked because if a resize is necessary, then all of the memory in the data structure

may need to be copied to a different location. On the other hand, in a linked list, only the node which is being inserted and the nodes between which the new node is being inserted need to be locked. Thus, in a linked-list implemented with multi-threading in mind, one thread should be able to read from the beginning of the list and another thread should be able to write to to the end of the list at the same exact time without issues. I believe the linked list approach likely improves performance, although I have not tested it. It is also worth investigating if linked list performance noticeably degrades in multi-threaded programs because of the mutex access during malloc.

Linked lists seem to pop up fairly often in early CS education topics despite their fairly limited use in further education. I think part of the reason for this is to more easily introduce the idea of data connected by pointers or in some way "linked" such that the hop to more advanced data structures, like trees, is more natural.