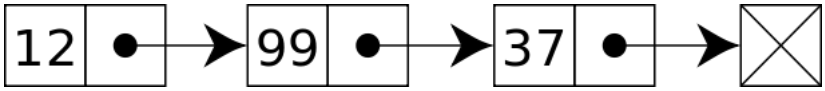
Linked list

From Wikipedia, the free encyclopedia

In computer science, a **linked list** is a linear collection of data elements, called nodes, each pointing to the next node by means of a pointer. It is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of data and a reference (in other words, a *link*) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration. More complex variants add additional links, allowing efficient insertion or removal from arbitrary element references.



A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list.

Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data types, including lists (the abstract data type), stacks, queues, associative arrays, and S-expressions, though it is not uncommon to implement the other data structures directly without using a list as the basis of implementation.

The principal benefit of a linked list over a conventional array is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk, while an array has to be declared in the source code, before compiling and running the program. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

On the other hand, simple linked lists by themselves do not allow random access to the data, or any form of efficient indexing. Thus, many basic operations — such as obtaining the last node of the list (assuming that the last node is not maintained as separate node reference in the list structure), or finding a node that contains a given datum, or locating the place where a new node should be inserted — may require sequential scanning of most or all of the list elements. The advantages and disadvantages of using linked lists are given below.

Contents

- 1 Advantages
- 2 Disadvantages
- 3 History
- 4 Basic concepts and nomenclature
  - 4.1 Singly linked list
  - 4.2 Doubly linked list
  - 4.3 Multiply linked list
  - 4.4 Circular Linked list
  - 4.5 Sentinel nodes
  - 4.6 Empty lists
  - 4.7 Hash linking
  - 4.8 List handles
  - 4.9 Combining alternatives
- 5 Tradeoffs
  - 5.1 Linked lists vs. dynamic arrays

- 5.2 Singly linked linear lists vs. other lists
- 5.3 Doubly linked vs. singly linked
- 5.4 Circularly linked vs. linearly linked
- 5.5 Using sentinel nodes
- 6 Linked list operations
  - 6.1 Linearly linked lists
    - 6.1.1 Singly linked lists
  - 6.2 Circularly linked list
    - 6.2.1 Algorithms
- 7 Linked lists using arrays of nodes
- 8 Language support
- 9 Internal and external storage
  - 9.1 Example of internal and external storage
  - 9.2 Speeding up search
  - 9.3 Random access lists
- 10 Related data structures
- 11 Notes
- 12 Footnotes
- 13 References
- 14 External links

Advantages

- Linked lists are a dynamic data structure, which can grow and be pruned, allocating and deallocating memory while the program is running.
- Insertion and deletion node operations are easily implemented in a linked list.
- Dynamic data structures such as stacks and queues can be implemented using a linked list.
- There is no need to define an initial size for a Linked list.
- Items can be added or removed from the middle of list.

Disadvantages

- They use more memory than arrays because of the storage used by their pointers.
- Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequential access.
- Nodes are stored incontiguously, greatly increasing the time required to access individual elements within the list, especially with a CPU cache.
- Difficulties arise in linked lists when it comes to reverse traversing. For instance, singly linked lists are cumbersome to navigate backwards[1] and while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a back-pointer.

History

Linked lists were developed in 1955–1956 by Allen Newell, Cliff Shaw and Herbert A. Simon at RAND Corporation as the primary data structure for their Information Processing Language. IPL was used by the authors to develop several early artificial intelligence programs, including the Logic Theory Machine, the General Problem Solver, and a computer chess program. Reports on their work appeared in IRE Transactions on Information Theory in 1956, and several conference proceedings from 1957 to 1959, including Proceedings of the Western Joint Computer Conference in 1957 and 1958, and Information Processing (Proceedings of the first UNESCO International Conference on Information Processing) in 1959. The now-classic diagram consisting of blocks

representing list nodes with arrows pointing to successive list nodes appears in "Programming the Logic Theory Machine" by Newell and Shaw in Proc. WJCC, February 1957. Newell and Simon were recognized with the ACM Turing Award in 1975 for having "made basic contributions to artificial intelligence, the psychology of human cognition, and list processing". The problem of machine translation for natural language processing led Victor Yngve at Massachusetts Institute of Technology (MIT) to use linked lists as data structures in his COMIT programming language for computer research in the field of linguistics. A report on this language entitled "A programming language for mechanical translation" appeared in Mechanical Translation in 1958.

LISP, standing for list processor, was created by John McCarthy in 1958 while he was at MIT and in 1960 he published its design in a paper in the Communications of the ACM, entitled "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". One of LISP's major data structures is the linked list.

By the early 1960s, the utility of both linked lists and languages which use these structures as their primary data representation was well established. Bert Green of the MIT Lincoln Laboratory published a review article entitled "Computer languages for symbol manipulation" in IRE Transactions on Human Factors in Electronics in March 1961 which summarized the advantages of the linked list approach. A later review article, "A Comparison of list-processing computer languages" by Bobrow and Raphael, appeared in Communications of the ACM in April 1964.

Several operating systems developed by Technical Systems Consultants (originally of West Lafayette Indiana, and later of Chapel Hill, North Carolina) used singly linked lists as file structures. A directory entry pointed to the first sector of a file, and succeeding portions of the file were located by traversing pointers. Systems using this technique included Flex (for the Motorola 6800 CPU), mini-Flex (same CPU), and Flex9 (for the Motorola 6809 CPU). A variant developed by TSC for and marketed by Smoke Signal Broadcasting in California, used doubly linked lists in the same manner.

The TSS/360 operating system, developed by IBM for the System 360/370 machines, used a double linked list for their file system catalog. The directory structure was similar to Unix, where a directory could contain files and other directories and extend to any depth.

Basic concepts and nomenclature

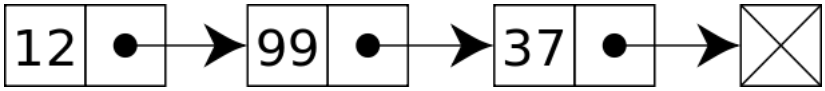
Each record of a linked list is often called an 'element' or 'node'.

The field of each node that contains the address of the next node is usually called the 'next link' or 'next pointer'. The remaining fields are known as the 'data', 'information', 'value', 'cargo', or 'payload' fields.

The 'head' of a list is its first node. The 'tail' of a list may refer either to the rest of the list after the head, or to the last node in the list. In Lisp and some derived languages, the next node may be called the 'cdr' (pronounced *could-er*) of the list, while the payload of the head node may be called the 'car'.

Singly linked list

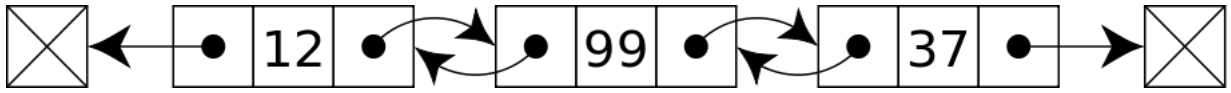
Singly linked lists contain nodes which have a data field as well as a 'next' field, which points to the next node in line of nodes. Operations that can be performed on singly linked lists include insertion, deletion and traversal.



A singly linked list whose nodes contain two fields: an integer value and a link to the next node

Doubly linked list

In a 'doubly linked list', each node contains, besides the next-node link, a second link field pointing to the 'previous' node in the sequence. The two links may be called 'forward('s') and 'backwards', or 'next' and 'prev'('previous').



A doubly linked list whose nodes contain three fields: an integer value, the link forward to the next node, and the link backward to the previous node

A technique known as XOR-linking allows a doubly linked list to be implemented using a single link field in each node. However, this technique requires the ability to do bit operations on addresses, and therefore may not be available in some high-level languages.

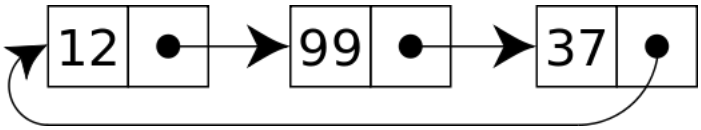
Many modern operating systems use doubly linked lists to maintain references to active processes, threads, and other dynamic objects.[2] A common strategy for rootkits to evade detection is to unlink themselves from these lists.[3]

Multiply linked list

In a 'multiply linked list', each node contains two or more link fields, each field being used to connect the same set of data records in a different order (e.g., by name, by department, by date of birth, etc.). While doubly linked lists can be seen as special cases of multiply linked list, the fact that the two orders are opposite to each other leads to simpler and more efficient algorithms, so they are usually treated as a separate case.

Circular Linked list

In the last node of a list, the link field often contains a null reference, a special value used to indicate the lack of further nodes. A less common convention is to make it point to the first node of the list; in that case the list is said to be 'circular' or 'circularly linked'; otherwise it is said to be 'open' or 'linear'.



A circular linked list

In the case of a circular doubly linked list, the only change that occurs is that the end, or "tail", of the said list is linked back to the front, or "head", of the list and vice versa.

Sentinel nodes

In some implementations an extra 'sentinel' or 'dummy' node may be added before the first data record or after the last one. This convention simplifies and accelerates some list-handling algorithms, by ensuring that all links can be safely dereferenced and that every list (even one that contains no data elements) always has a "first" and "last" node.

Empty lists

An empty list is a list that contains no data records. This is usually the same as saying that it has zero nodes. If sentinel nodes are being used, the list is usually said to be empty when it has only sentinel nodes.

Hash linking

The link fields need not be physically part of the nodes. If the data records are stored in an array and referenced by their indices, the link field may be stored in a separate array with the same indices as the data records.

List handles

Since a reference to the first node gives access to the whole list, that reference is often called the 'address', 'pointer', or 'handle' of the list. Algorithms that manipulate linked lists usually get such handles to the input lists and return the handles to the resulting lists. In fact, in the context of such algorithms, the word "list" often means "list handle". In some situations, however, it may be convenient to refer to a list by a handle that consists of two links, pointing to its first and last nodes.

Combining alternatives

The alternatives listed above may be arbitrarily combined in almost every way, so one may have circular doubly linked lists without sentinels, circular singly linked lists with sentinels, etc.

Tradeoffs

As with most choices in computer programming and design, no method is well suited to all circumstances. A linked list data structure might work well in one case, but cause problems in another. This is a list of some of the common tradeoffs involving linked list structures.

Linked lists vs. dynamic arrays

Comparison of list data structures

	Linked list	Array	Dynamic array	Balanced tree	Random access list	hashed array tree
Indexing	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$[4]	$\Theta(1)$
Insert/delete at beginning	$\Theta(1)$	N/A	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(n)$
Insert/delete at end	$\Theta(n)$ when last element is unknown; $\Theta(1)$ when last element is known	N/A	$\Theta(1)$ amortized	$\Theta(\log n)$	$\Theta(\log n)$ updating	$\Theta(1)$ amortized
Insert/delete in middle	search time + $\Theta(1)$[5][6][7]	N/A	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$ updating	$\Theta(n)$
Wasted space (average)	$\Theta(n)$	0	$\Theta(n)$[8]	$\Theta(n)$	$\Theta(n)$	$\Theta(\sqrt{n})$

A *dynamic array* is a data structure that allocates all elements contiguously in memory, and keeps a count of the current number of elements. If the space reserved for the dynamic array is exceeded, it is reallocated and (possibly) copied, which is an expensive operation.

Linked lists have several advantages over dynamic arrays. Insertion or deletion of an element at a specific point of a list, assuming that we have indexed a pointer to the node (before the one to be removed, or before the insertion point) already, is a constant-time operation (otherwise without this reference it is O(n)), whereas insertion in a

dynamic array at random locations will require moving half of the elements on average, and all the elements in the worst case. While one can "delete" an element from an array in constant time by somehow marking its slot as "vacant", this causes fragmentation that impedes the performance of iteration.

Moreover, arbitrarily many elements may be inserted into a linked list, limited only by the total memory available; while a dynamic array will eventually fill up its underlying array data structure and will have to reallocate — an expensive operation, one that may not even be possible if memory is fragmented, although the cost of reallocation can be averaged over insertions, and the cost of an insertion due to reallocation would still be amortized O(1). This helps with appending elements at the array's end, but inserting into (or removing from) middle positions still carries prohibitive costs due to data moving to maintain contiguity. An array from which many elements are removed may also have to be resized in order to avoid wasting too much space.

On the other hand, dynamic arrays (as well as fixed-size array data structures) allow constant-time random access, while linked lists allow only sequential access to elements. Singly linked lists, in fact, can be easily traversed in only one direction. This makes linked lists unsuitable for applications where it's useful to look up an element by its index quickly, such as heapsort. Sequential access on arrays and dynamic arrays is also faster than on linked lists on many machines, because they have optimal locality of reference and thus make good use of data caching.

Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or boolean values, because the storage overhead for the links may exceed by a factor of two or more the size of the data. In contrast, a dynamic array requires only the space for the data itself (and a very small amount of control data).[note 1] It can also be slow, and with a naïve allocator, wasteful, to allocate memory separately for each new element, a problem generally solved using memory pools.

Some hybrid solutions try to combine the advantages of the two representations. Unrolled linked lists store several elements in each list node, increasing cache performance while decreasing memory overhead for references. CDR coding does both these as well, by replacing references with the actual data referenced, which extends off the end of the referencing record.

A good example that highlights the pros and cons of using dynamic arrays vs. linked lists is by implementing a program that resolves the Josephus problem. The Josephus problem is an election method that works by having a group of people stand in a circle. Starting at a predetermined person, you count around the circle *n* times. Once you reach the *n*th person, take them out of the circle and have the members close the circle. Then count around the circle the same *n* times and repeat the process, until only one person is left. That person wins the election. This shows the strengths and weaknesses of a linked list vs. a dynamic array, because if you view the people as connected nodes in a circular linked list then it shows how easily the linked list is able to delete nodes (as it only has to rearrange the links to the different nodes). However, the linked list will be poor at finding the next person to remove and will need to search through the list until it finds that person. A dynamic array, on the other hand, will be poor at deleting nodes (or elements) as it cannot remove one node without individually shifting all the elements up the list by one. However, it is exceptionally easy to find the *n*th person in the circle by directly referencing them by their position in the array.

The list ranking problem concerns the efficient conversion of a linked list representation into an array. Although trivial for a conventional computer, solving this problem by a parallel algorithm is complicated and has been the subject of much research.

A balanced tree has similar memory access patterns and space overhead to a linked list while permitting much more efficient indexing, taking O(log n) time instead of O(n) for a random access. However, insertion and deletion operations are more expensive due to the overhead of tree manipulations to maintain balance. Schemes exist for trees to automatically maintain themselves in a balanced state: AVL trees or red-black trees.

Singly linked linear lists vs. other lists

While doubly linked and circular lists have advantages over singly linked linear lists, linear lists offer some advantages that make them preferable in some situations.

A singly linked linear list is a recursive data structure, because it contains a pointer to a *smaller* object of the same type. For that reason, many operations on singly linked linear lists (such as merging two lists, or enumerating the elements in reverse order) often have very simple recursive algorithms, much simpler than any solution using iterative commands. While those recursive solutions can be adapted for doubly linked and circularly linked lists, the procedures generally need extra arguments and more complicated base cases.

Linear singly linked lists also allow tail-sharing, the use of a common final portion of sub-list as the terminal portion of two different lists. In particular, if a new node is added at the beginning of a list, the former list remains available as the tail of the new one — a simple example of a persistent data structure. Again, this is not true with the other variants: a node may never belong to two different circular or doubly linked lists.

In particular, end-sentinel nodes can be shared among singly linked non-circular lists. The same end-sentinel node may be used for *every* such list. In Lisp, for example, every proper list ends with a link to a special node, denoted by `nil` or `()`, whose `CAR` and `CDR` links point to itself. Thus a Lisp procedure can safely take the `CAR` or `CDR` of *any* list.

The advantages of the fancy variants are often limited to the complexity of the algorithms, not in their efficiency. A circular list, in particular, can usually be emulated by a linear list together with two variables that point to the first and last nodes, at no extra cost.

Doubly linked vs. singly linked

Double-linked lists require more space per node (unless one uses XOR-linking), and their elementary operations are more expensive; but they are often easier to manipulate because they allow fast and easy sequential access to the list in both directions. In a doubly linked list, one can insert or delete a node in a constant number of operations given only that node's address. To do the same in a singly linked list, one must have the *address of the pointer* to that node, which is either the handle for the whole list (in case of the first node) or the link field in the *previous* node. Some algorithms require access in both directions. On the other hand, doubly linked lists do not allow tail-sharing and cannot be used as persistent data structures.

Circularly linked vs. linearly linked

A circularly linked list may be a natural option to represent arrays that are naturally circular, e.g. the corners of a polygon, a pool of buffers that are used and released in FIFO ("first in, first out") order, or a set of processes that should be time-shared in round-robin order. In these applications, a pointer to any node serves as a handle to the whole list.

With a circular list, a pointer to the last node gives easy access also to the first node, by following one link. Thus, in applications that require access to both ends of the list (e.g., in the implementation of a queue), a circular structure allows one to handle the structure by a single pointer, instead of two.

A circular list can be split into two circular lists, in constant time, by giving the addresses of the last node of each piece. The operation consists in swapping the contents of the link fields of those two nodes. Applying the same operation to any two nodes in two distinct lists joins the two list into one. This property greatly simplifies some algorithms and data structures, such as the quad-edge and face-edge.

The simplest representation for an empty *circular* list (when such a thing makes sense) is a null pointer, indicating that the list has no nodes. Without this choice, many algorithms have to test for this special case, and handle it separately. By contrast, the use of null to denote an empty *linear* list is more natural and often creates fewer special

cases.

Using sentinel nodes

Sentinel node may simplify certain list operations, by ensuring that the next or previous nodes exist for every element, and that even empty lists have at least one node. One may also use a sentinel node at the end of the list, with an appropriate data field, to eliminate some end-of-list tests. For example, when scanning the list looking for a node with a given value x, setting the sentinel's data field to x makes it unnecessary to test for end-of-list inside the loop. Another example is the merging two sorted lists: if their sentinels have data fields set to $+\infty$, the choice of the next output node does not need special handling for empty lists.

However, sentinel nodes use up extra space (especially in applications that use many short lists), and they may complicate other operations (such as the creation of a new empty list).

However, if the circular list is used merely to simulate a linear list, one may avoid some of this complexity by adding a single sentinel node to every list, between the last and the first data nodes. With this convention, an empty list consists of the sentinel node alone, pointing to itself via the next-node link. The list handle should then be a pointer to the last data node, before the sentinel, if the list is not empty; or to the sentinel itself, if the list is empty.

The same trick can be used to simplify the handling of a doubly linked linear list, by turning it into a circular doubly linked list with a single sentinel node. However, in this case, the handle should be a single pointer to the dummy node itself.[9]

Linked list operations

When manipulating linked lists in-place, care must be taken to not use values that you have invalidated in previous assignments. This makes algorithms for inserting or deleting linked list nodes somewhat subtle. This section gives pseudocode for adding or removing nodes from singly, doubly, and circularly linked lists in-place. Throughout we will use *null* to refer to an end-of-list marker or sentinel, which may be implemented in a number of ways.

Linearly linked lists

Singly linked lists

Our node data structure will have two fields. We also keep a variable *firstNode* which always points to the first node in the list, or is *null* for an empty list.

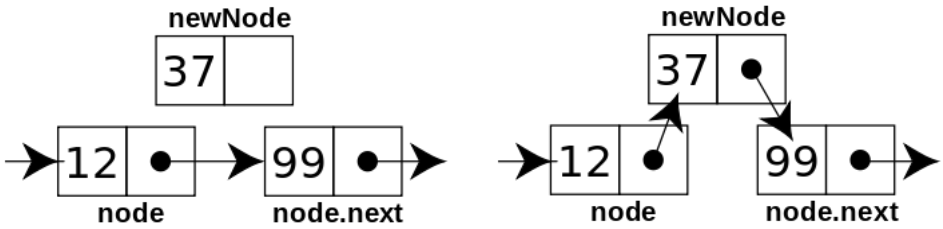
```
record Node
{
   data; // The data being stored in the node
   Node next // A reference to the next node, null for last node
}
```

```
record List
{
    Node firstNode // points to first node of list; null for empty list
}
```

Traversal of a singly linked list is simple, beginning at the first node and following each *next* link until we come to the end:

```
node := list.firstNode
while node not null
    (do something with node.data)
    node := node.next
```

The following code inserts a node after an existing node in a singly linked list. The diagram shows how it works. Inserting a node before an existing one cannot be done directly; instead, one must keep track of the previous node and insert a node after it.

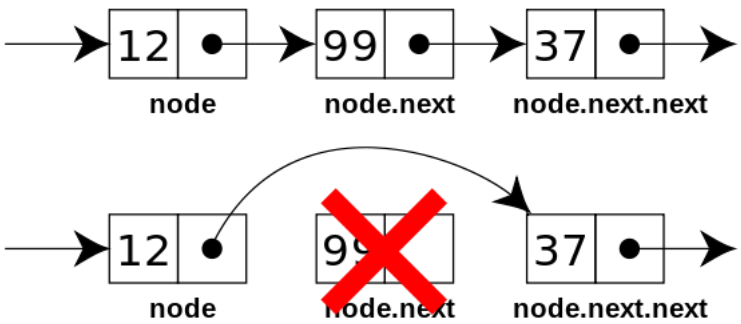



```
function insertAfter(Node node, Node newNode) // insert newNode after node
    newNode.next := node.next
    node.next    := newNode
```

Inserting at the beginning of the list requires a separate function. This requires updating *firstNode*.

```
function insertBeginning(List list, Node newNode) // insert node before current first node
    newNode.next   := list.firstNode
    list.firstNode := newNode
```

Similarly, we have functions for removing the node *after* a given node, and for removing a node from the beginning of the list. The diagram demonstrates the former. To find and remove a particular node, one must again keep track of the previous element.




```
function removeAfter(Node node) // remove node past this one
    obsoleteNode := node.next
    node.next := node.next.next
    destroy obsoleteNode
```

```
function removeBeginning(List list) // remove first node
    obsoleteNode := list.firstNode
    list.firstNode := list.firstNode.next // point past deleted node
    destroy obsoleteNode
```

Notice that `removeBeginning()` sets `list.firstNode` to `null` when removing the last node in the list.

Since we can't iterate backwards, efficient `insertBefore` or `removeBefore` operations are not possible. Inserting to a list before a specific node requires traversing the list, which would have a worst case running time of O(n).

Appending one linked list to another can be inefficient unless a reference to the tail is kept as part of the List structure, because we must traverse the entire first list in order to find the tail, and then append the second list to this. Thus, if two linearly linked lists are each of length n, list appending has asymptotic time complexity of $O(n)$. In the Lisp family of languages, list appending is provided by the `append` procedure.

Many of the special cases of linked list operations can be eliminated by including a dummy element at the front of the list. This ensures that there are no special cases for the beginning of the list and renders both `insertBeginning()` and `removeBeginning()` unnecessary. In this case, the first useful data in the list will be found at `list.`**`firstNode`**`.next`.

Circularly linked list

In a circularly linked list, all nodes are linked in a continuous circle, without using *null*. For lists with a front and a back (such as a queue), one stores a reference to the last node in the list. The *next* node after the last node is the first node. Elements can be added to the back of the list and removed from the front in constant time.

Circularly linked lists can be either singly or doubly linked.

Both types of circularly linked lists benefit from the ability to traverse the full list beginning at any given node. This often allows us to avoid storing *firstNode* and *lastNode*, although if the list may be empty we need a special representation for the empty list, such as a *lastNode* variable which points to some node in the list or is *null* if it's empty; we use such a *lastNode* here. This representation significantly simplifies adding and removing nodes with a non-empty list, but empty lists are then a special case.

Algorithms

Assuming that *someNode* is some node in a non-empty circular singly linked list, this code iterates through that list starting with *someNode*:

```
function iterate(someNode)
  if someNode ≠ null
    node := someNode
    do
      do something with node.value
      node := node.next
    while node ≠ someNode
```

Notice that the test "**while** node ≠ someNode" must be at the end of the loop. If the test was moved to the beginning of the loop, the procedure would fail whenever the list had only one node.

This function inserts a node "newNode" into a circular linked list after a given node "node". If "node" is null, it assumes that the list is empty.

```
function insertAfter(Node node, Node newNode)
    if node = null
      newNode.next := newNode
    else
      newNode.next := node.next
      node.next := newNode
```

Suppose that "L" is a variable pointing to the last node of a circular linked list (or null if the list is empty). To append "newNode" to the *end* of the list, one may do

```
insertAfter(L, newNode)
L := newNode
```

To insert "newNode" at the *beginning* of the list, one may do

```
insertAfter(L, newNode)
if L = null
  L := newNode
```

Linked lists using arrays of nodes

Languages that do not support any type of reference can still create links by replacing pointers with array indices. The approach is to keep an array of records, where each record has integer fields indicating the index of the next (and possibly previous) node in the array. Not all nodes in the array need be used. If records are also not supported, parallel arrays can often be used instead.

As an example, consider the following linked list record that uses arrays instead of pointers:

```
record Entry {
   integer next; // index of next entry in array
   integer prev; // previous entry (if double-linked)
   string name;
   real balance;
}
```

A linked list can be built by creating an array of these structures, and an integer variable to store the index of the first element.

```
integer listHead
Entry Records[1000]
```

Links between elements are formed by placing the array index of the next (or previous) cell into the Next or Prev field within a given element. For example:

Index	Next	Prev	Name	Balance
0	1	4	Jones, John	123.45
1	−1	0	Smith, Joseph	234.56
2 (listHead)	4	−1	Adams, Adam	0.00
3			Ignore, Ignatius	999.99
4	0	2	Another, Anita	876.54
5				
6				
7				

In the above example, `ListHead` would be set to 2, the location of the first entry in the list. Notice that entry 3 and 5 through 7 are not part of the list. These cells are available for any additions to the list. By creating a `ListFree` integer variable, a free list could be created to keep track of what cells are available. If all entries are in use, the size of the array would have to be increased or some elements would have to be deleted before new entries could be stored in the list.

The following code would traverse the list and display names and account balance:

```
i := listHead
while i ≥ 0 // loop through the list
    print i, Records[i].name, Records[i].balance // print entry
    i := Records[i].next
```

When faced with a choice, the advantages of this approach include:

- The linked list is relocatable, meaning it can be moved about in memory at will, and it can also be quickly and directly serialized for storage on disk or transfer over a network.
- Especially for a small list, array indexes can occupy significantly less space than a full pointer on many architectures.
- Locality of reference can be improved by keeping the nodes together in memory and by periodically rearranging them, although this can also be done in a general store.
- Naïve dynamic memory allocators can produce an excessive amount of overhead storage for each node allocated; almost no allocation overhead is incurred per node in this approach.
- Seizing an entry from a pre-allocated array is faster than using dynamic memory allocation for each node, since dynamic memory allocation typically requires a search for a free memory block of the desired size.

This approach has one main disadvantage, however: it creates and manages a private memory space for its nodes. This leads to the following issues:

- It increases complexity of the implementation.
- Growing a large array when it is full may be difficult or impossible, whereas finding space for a new linked list node in a large, general memory pool may be easier.
- Adding elements to a dynamic array will occasionally (when it is full) unexpectedly take linear (O(n)) instead of constant time (although it's still an amortized constant).
- Using a general memory pool leaves more memory for other data if the list is smaller than expected or if many nodes are freed.

For these reasons, this approach is mainly used for languages that do not support dynamic memory allocation. These disadvantages are also mitigated if the maximum size of the list is known at the time the array is created.

Language support

Many programming languages such as Lisp and Scheme have singly linked lists built in. In many functional languages, these lists are constructed from nodes, each called a *cons* or *cons cell*. The cons has two fields: the *car*, a reference to the data for that node, and the *cdr*, a reference to the next node. Although cons cells can be used to build other data structures, this is their primary purpose.

In languages that support abstract data types or templates, linked list ADTs or templates are available for building linked lists. In other languages, linked lists are typically built using references together with records.

Internal and external storage

When constructing a linked list, one is faced with the choice of whether to store the data of the list directly in the linked list nodes, called *internal storage*, or merely to store a reference to the data, called *external storage*. Internal storage has the advantage of making access to the data more efficient, requiring less storage overall, having better locality of reference, and simplifying memory management for the list (its data is allocated and deallocated at the same time as the list nodes).

External storage, on the other hand, has the advantage of being more generic, in that the same data structure and machine code can be used for a linked list no matter what the size of the data is. It also makes it easy to place the same data in multiple linked lists. Although with internal storage the same data can be placed in multiple lists by including multiple *next* references in the node data structure, it would then be necessary to create separate routines to add or delete cells based on each field. It is possible to create additional linked lists of elements that use internal storage by using external storage, and having the cells of the additional linked lists store references to the nodes of the linked list containing the data.

In general, if a set of data structures needs to be included in linked lists, external storage is the best approach. If a set of data structures need to be included in only one linked list, then internal storage is slightly better, unless a generic linked list package using external storage is available. Likewise, if different sets of data that can be stored in the same data structure are to be included in a single linked list, then internal storage would be fine.

Another approach that can be used with some languages involves having different data structures, but all have the initial fields, including the *next* (and *prev* if double linked list) references in the same location. After defining separate structures for each type of data, a generic structure can be defined that contains the minimum amount of data shared by all the other structures and contained at the top (beginning) of the structures. Then generic routines can be created that use the minimal structure to perform linked list type operations, but separate routines can then handle the specific data. This approach is often used in message parsing routines, where several types of messages are received, but all start with the same set of fields, usually including a field for message type. The generic routines are used to add new messages to a queue when they are received, and remove them from the queue in order to process the message. The message type field is then used to call the correct routine to process the specific type of message.

Example of internal and external storage

Suppose you wanted to create a linked list of families and their members. Using internal storage, the structure might look like the following:

```
record member { // member of a family
    member next;
    string firstName;
    integer age;
}
record family { // the family itself
    family next;
    string lastName;
    string address;
    member members // head of list of members of this family
}
```

To print a complete list of families and their members using internal storage, we could write:

```
aFamily := Families // start at head of families list
while aFamily ≠ null // loop through list of families
    print information about family
    aMember := aFamily.members // get head of list of this family's members
    while aMember ≠ null // loop through list of members
        print information about member
```

```
        aMember := aMember.next
    aFamily := aFamily.next
```

Using external storage, we would create the following structures:

```
record node { // generic link structure
    node next;
    pointer data // generic pointer for data at node
}
record member { // structure for family member
    string firstName;
    integer age
}
record family { // structure for family
    string lastName;
    string address;
    node members // head of list of members of this family
}
```

To print a complete list of families and their members using external storage, we could write:

```
famNode := Families // start at head of families list
while famNode ≠ null // loop through list of families
    aFamily := (family) famNode.data // extract family from node
    print information about family
    memNode := aFamily.members // get list of family members
    while memNode ≠ null // loop through list of members
        aMember := (member)memNode.data // extract member from node
        print information about member
        memNode := memNode.next
    famNode := famNode.next
```

Notice that when using external storage, an extra step is needed to extract the record from the node and cast it into the proper data type. This is because both the list of families and the list of members within the family are stored in two linked lists using the same data structure (*node*), and this language does not have parametric types.

As long as the number of families that a member can belong to is known at compile time, internal storage works fine. If, however, a member needed to be included in an arbitrary number of families, with the specific number known only at run time, external storage would be necessary.

Speeding up search

Finding a specific element in a linked list, even if it is sorted, normally requires O(n) time (linear search). This is one of the primary disadvantages of linked lists over other data structures. In addition to the variants discussed above, below are two simple ways to improve search time.

In an unordered list, one simple heuristic for decreasing average search time is the *move-to-front heuristic*, which simply moves an element to the beginning of the list once it is found. This scheme, handy for creating simple caches, ensures that the most recently used items are also the quickest to find again.

Another common approach is to "index" a linked list using a more efficient external data structure. For example, one can build a red-black tree or hash table whose elements are references to the linked list nodes. Multiple such indexes can be built on a single list. The disadvantage is that these indexes may need to be updated each time a node is added or removed (or at least, before that index is used again).

Random access lists

A random access list is a list with support for fast random access to read or modify any element in the list.[10] One possible implementation is a skew binary random access list using the skew binary number system, which involves a list of trees with special properties; this allows worst-case constant time head/cons operations, and worst-case logarithmic time random access to an element by index.[10] Random access lists can be implemented as persistent data structures.[10]

Random access lists can be viewed as immutable linked lists in that they likewise support the same O(1) head and tail operations.[10]

A simple extension to random access lists is the min-list, which provides an additional operation that yields the minimum element in the entire list in constant time (without mutation complexities).[10]

Related data structures

Both stacks and queues are often implemented using linked lists, and simply restrict the type of operations which are supported.

The skip list is a linked list augmented with layers of pointers for quickly jumping over large numbers of elements, and then descending to the next layer. This process continues down to the bottom layer, which is the actual list.

A binary tree can be seen as a type of linked list where the elements are themselves linked lists of the same nature. The result is that each node may include a reference to the first node of one or two other linked lists, which, together with their contents, form the subtrees below that node.

An unrolled linked list is a linked list in which each node contains an array of data values. This leads to improved cache performance, since more list elements are contiguous in memory, and reduced memory overhead, because less metadata needs to be stored for each element of the list.

A hash table may use linked lists to store the chains of items that hash to the same position in the hash table.

A heap shares some of the ordering properties of a linked list, but is almost always implemented using an array. Instead of references from node to node, the next and previous data indexes are calculated using the current data's index.

A self-organizing list rearranges its nodes based on some heuristic which reduces search times for data retrieval by keeping commonly accessed nodes at the head of the list.

Notes

1. The amount of control data required for a dynamic array is usually of the form $K + B * n$, where K is a per-array constant, B is a per-dimension constant, and n is the number of dimensions. K and B are typically on the order of 10 bytes.

Footnotes

1. Skiena, Steven S. (2009). *The Algorithm Design Manual* (2nd ed.). Springer. p. 76. ISBN 9781848000704. "We can do nothing without this list predecessor, and so must spend linear time searching for it on a singly-linked list."
2. http://www.osronline.com/article.cfm?article=499
3. http://www.cs.dartmouth.edu/~sergey/me/cs/cs108/rootkits/bh-us-04-butler.pdf

4. Chris Okasaki (1995). "Purely Functional Random-Access Lists". *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*: 86–95. doi:10.1145/224164.224187.
5. Gerald Kruse. CS 240 Lecture Notes (http://www.juniata.edu/faculty/kruse/cs240/syllabus.htm): Linked Lists Plus: Complexity Trade-offs (http://www.juniata.edu/faculty/kruse/cs240/linkedlist2.htm). Juniata College. Spring 2008.
6. *Day 1 Keynote - Bjarne Stroustrup: C++11 Style* (http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style) at *GoingNative 2012* on *channel9.msdn.com* from minute 45 or foil 44
7. *Number crunching: Why you should never, ever, EVER use linked-list in your code again* (http://kjellkod.wordpress.com/2012/02/25/why-you-should-never-ever-ever-use-linked-list-in-your-code-again/) at *kjellkod.wordpress.com*
8. Brodnik, Andrej; Carlsson, Svante; Sedgewick, Robert; Munro, JI; Demaine, ED (1999), *Resizable Arrays in Optimal Time and Space (Technical Report CS-99-09)* (PDF), Department of Computer Science, University of Waterloo
9. Ford, William; Topp, William (2002). *Data Structures with C++ using STL* (Second ed.). Prentice-Hall. pp. 466–467. ISBN 0-13-085850-1.
10. Okasaki, Chris (1995). *Purely Functional Random-Access Lists* (PS). *In Functional Programming Languages and Computer Architecture*. ACM Press. pp. 86–95. Retrieved May 7, 2015.

References

- Juan, Angel (2006). "Ch20 –Data Structures; ID06 - PROGRAMMING with JAVA (slide part of the book 'Big Java', by CayS. Horstmann)" (PDF). p. 3.
- Black, Paul E. (2004-08-16). Pieterse, Vreda; Black, Paul E., eds. "linked list". *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology. Retrieved 2004-12-14.
- Antonakos, James L.; Mansfield, Kenneth C., Jr. (1999). *Practical Data Structures Using C/C++*. Prentice-Hall. pp. 165–190. ISBN 0-13-280843-9.
- Collins, William J. (2005) [2002]. *Data Structures and the Java Collections Framework*. New York: McGraw Hill. pp. 239–303. ISBN 0-07-282379-8.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2003). *Introduction to Algorithms*. MIT Press. pp. 205–213, 501–505. ISBN 0-262-03293-7.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "10.2: Linked lists". *Introduction to Algorithms* (2nd ed.). MIT Press. pp. 204–209. ISBN 0-262-03293-7.
- Green, Bert F., Jr. (1961). "Computer Languages for Symbol Manipulation". *IRE Transactions on Human Factors in Electronics* (2): 3–8. doi:10.1109/THFE2.1961.4503292.
- McCarthy, John (1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". *Communications of the ACM*. **3** (4): 184. doi:10.1145/367177.367199.
- Knuth, Donald (1997). "2.2.3-2.2.5". *Fundamental Algorithms* (3rd ed.). Addison-Wesley. pp. 254–298. ISBN 0-201-89683-4.
- Newell, Allen; Shaw, F. C. (1957). "Programming the Logic Theory Machine". *Proceedings of the Western Joint Computer Conference*: 230–240.
- Parlante, Nick (2001). "Linked list basics" (PDF). Stanford University. Retrieved 2009-09-21.
- Sedgewick, Robert (1998). *Algorithms in C*. Addison Wesley. pp. 90–109. ISBN 0-201-31452-5.
- Shaffer, Clifford A. (1998). *A Practical Introduction to Data Structures and Algorithm Analysis*. New Jersey: Prentice Hall. pp. 77–102. ISBN 0-13-660911-2.
- Wilkes, Maurice Vincent (1964). "An Experiment with a Self-compiling Compiler for a Simple List-Processing Language". *Annual Review in Automatic Programming*. Pergamon Press. **4** (1): 1. doi:10.1016/0066-4138(64)90013-8.
- Wilkes, Maurice Vincent (1964). "Lists and Why They are Useful". *Proceeds of the ACM National Conference, Philadelphia 1964*. ACM (P–64): F1–1.
- Shanmugasundaram, Kulesh (2005-04-04). "Linux Kernel Linked List Explained". Retrieved 2009-09-21.

External links

- Description (http://nist.gov/dads/HTML/linkedList.html) from the Dictionary of Algorithms and Data Structures

Wikimedia Commons has media related to ***Linked lists***.

- Introduction to Linked Lists (http://cslibrary.stanford.edu/103/), Stanford University Computer Science Library
- Linked List Problems (http://cslibrary.stanford.edu/105/), Stanford University Computer Science Library
- Open Data Structures - Chapter 3 - Linked Lists (http://opendatastructures.org/versions/edition-0.1g/ods-python/3_Linked_Lists.html)
- Patent for the idea of having nodes which are in several linked lists simultaneously (http://www.google.com/patents?vid=USPAT7028023) (note that this technique was widely used for many decades before the patent was granted)