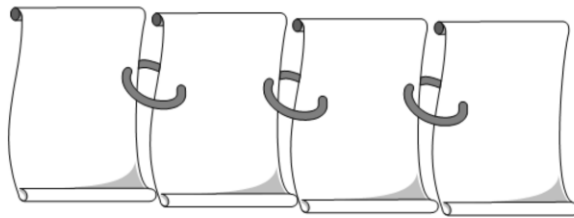


## Linked Lists

- **Objectives**
  - Study a different way to structure collections of data
- **Contents**
  - Printing and searching linked lists
  - Building linked lists
  - Deleting from a linked list
  - Doubly linked lists
- **Practical**
- **Summary**



---

The objective of this chapter is to describe one of the most common data structures in the computing industry. Linked lists are described in books on design strategy and in books on implementation techniques in many languages.

The chapter shows how code to manipulate linked lists may be easily written in C. It shows how recursive functions may be used to traverse list in forward or reverse order. The chapter then goes on to describe doubly linked lists, in which each node contains a pointer to the next and previous nodes in the list.

## Introduction

- **A significant problem with arrays in C is that elements must be contiguous in memory - there must be a single block of memory large enough to hold the entire array**
- **When new values are added into a sorted array, the array must be resorted - this is time consuming**
- **Linked lists allows potentially large data structures to be broken up into many smaller chunks and chained together**
- **To add a new value into a sorted list, the insertion position is determined and pointers changed - no time consuming moving of data is required**
- **A linked list is a collection of nodes connected by links (pointers)**
- **The first node is designated as the head of the list**
- **The last node usually contains a NULL pointer**
- **The list may be (easily) traversed in one direction**

Previously, we saw that the elements of an array must be placed next to one another in memory. This is to ensure that if a pointer is set to a particular element and incremented, it will point to the next element. When decremented, the pointer will move back to the previous element. This strategy would simply not work if there were holes in the array. Thus there are no holes. Unfortunately, it can be painfully slow to add a new element into a sorted array, because the array needs to be resorted after the insertion, and it is wasteful with memory.

The linked list is a collection of structures chained together (each node in the list contains a pointer to the next node). In the case of a two-way list, each node will also contain a pointer to its previous node as well. As we shall see, two-way lists are often more convenient than one-way lists because it is easier to traverse a two-way list in both directions when necessary.

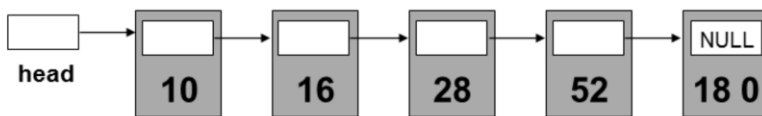
The differences between lists and arrays can be summarised as follows:

- The number of nodes in a list can increase and decrease dynamically; there is no fixed *array size* to worry about.
- Nodes can be inserted and removed without having to shuffle array elements about to compensate for the change in the number of items.
- Linked lists do not lend themselves to random access in the same way that arrays do. The only way to access an element in a linked list is to traverse the list sequentially from the beginning (or end) until the required item is reached. With arrays, the array elements occupy contiguous memory and can be accessed directly by index value.
- Without using a 'helper' array of pointers, linked lists cannot be used in conjunction with `bsearch()` and `qsort()` discussed previously.

## Representing nodes

- First we will consider singly linked lists

```
struct list_node
{
    int          data;
    struct list_node * next;
};
typedef struct list_node node;
```



The `struct` template shown above illustrates the simplest form of a linked list: a single, one-way list. Each node contains a pointer to the next node in the list, as well as the actual data for the node.

The list is manipulated through a single pointer `pHead` which points to the first node in the list. The end of the list is signified by a `NULL` pointer in the last node's `next` pointer. It is these two mechanisms which set the boundaries of the one-way list. A completely empty list will simply consist of a `NULL` value in the `pHead` pointer.

Each node in the list is allocated individually as and when needed using `malloc()` or an equivalent function. When a node is no longer required, it must be removed from the list and its memory must be relinquished using `free()`.

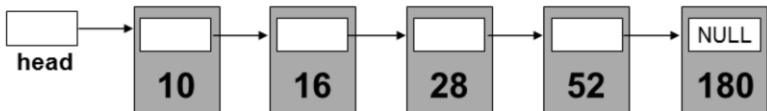
Note that the structure has been given a tag *and* we have used `typedef`. If we had written:

```
typedef struct list_node
{
    int    data;
    node *  next;      /* WRONG! */
} node;
```

it would not have compiled, since `node` is not available within the structure, only once it has been defined (i.e. after the `"}"`).

### Printing a linked list

- Consider a simple one-way linked list



- The list may be traversed as follows :

```
void print_list(node * p)
{
    while (p)
    {
        printf("%d\n", p->data);
        p++;
    }
}
```



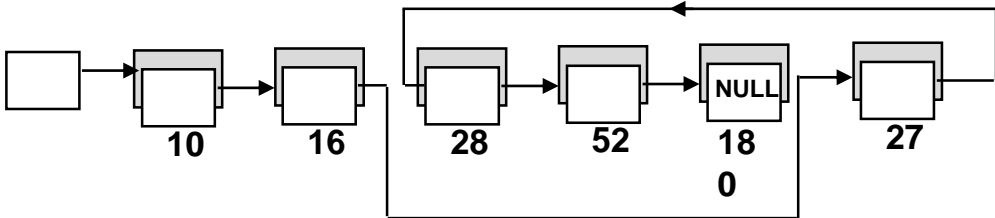
```
void print_list(node * p)
{
    while (p)
    {
        printf("%d\n", p->data);
        p = p->next;
    }
}
```



The diagram illustrates a simple one-way list with the data being a single int. Note that the last node in the list contains a NULL to signify the end of the list.

The `print_list()` function is one of the simplest algorithms for traversing a list. The algorithm uses a loop construct to visit each node in the list and then moves on to the next one.

The first version of `print_list()` is incorrect, since it assumes that the nodes in the list occupy adjacent addresses in memory. This is true only in the case of an array. The example above makes the nodes *look* as though they are adjacent, however, consider the case where we insert 27 into the list. The list will then look as follows:



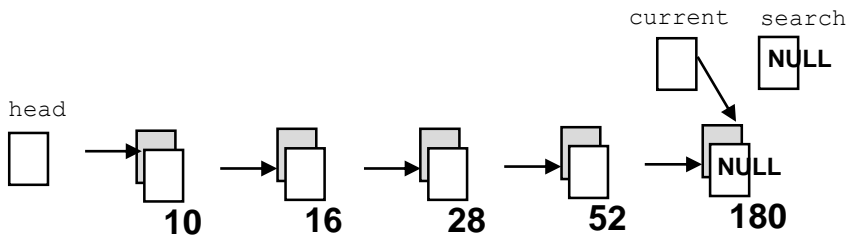
Now it is easier to see why a straightforward pointer increment will not visit the nodes in the correct order.

## Printing a linked list backwards

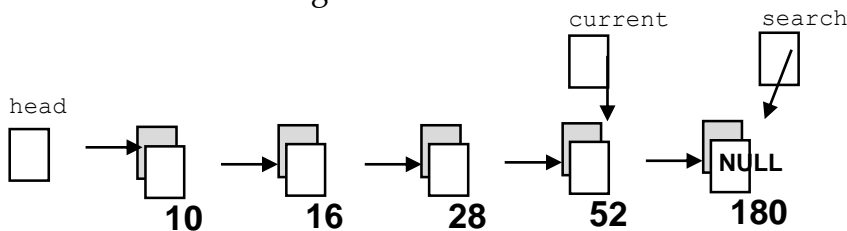
- Printing the list backwards is not quite as straightforward:

```
void print_list_backwards(node * head)
{
    node * current = head;
    node * search = 0;
    while (search != head)
    {
        current = head;
        while (current && current->next != search)
            current = current->next;
        if (current)
        {
            printf("%d", current->data);
            search = current;
        }
    }
    printf("\n");
}
```

Printing the list backwards is slightly more tricky. Firstly we set a pointer called `search` to `NULL`. This is the value we are going to search for in the `next` field of each node. We run along the list, pointing `current` to each of the nodes in turn. When `search` is found we must have reached the end of the list.



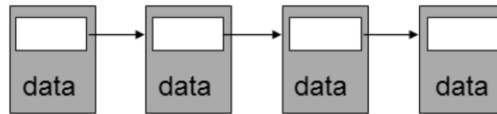
The next step is to search transfer `current` into `search` and look for the node which contains `search` in its `next` field. Unfortunately this involves starting at the front of the list and working our way forwards. Needless to say if the list were very long this would be time consuming.



We loop until we find ourselves searching for the head of the list.

## Recursion and linked lists

- A linked list is an inherently *recursive* data structure



- A linked list can be described as "a node followed by the remainder of the linked list"
- Recursive algorithms may usually be applied quite naturally to recursive data structures

Recursion is a subject shrouded in much myth and mysticism. In some circles it is frowned upon and claimed to be difficult to understand. In some it is discouraged, in others placed on a high altar and made the goal of all programming. Basically the whole idea comes down to defining a problem in terms of itself. For instance, consider the following definition of a linked list:

*A list is either empty, or it comprises a head (first item) and the tail (the **list** of remaining items)*

At first sight, this definition is not very helpful yet, looking more closely, a node is defined in terms of itself (a node is an item of data and a pointer to the next node).

The reason why recursion comes in for so much flack is that it requires stack space. When a function calls itself, stack space is required to hold the return address and the parameters. The more often a function calls itself the more stack space is required. Some languages do not use stacks. Early versions of FORTRAN for instance. Thus any form of recursion was banned. Other languages take recursion to heart. For instance in LISP it is difficult to do anything if you don't recurse first.

Recursion has become so popular that many modern day compilers have optimisers that will remove recursion. Thus you can write your code to be as recursive as possible, the compiler will write the code in an iterative way which will save you hitting your stack too hard.

Recursion removal is discussed in "*Algorithms in C*" by Robert Sedgewick, Addison-Wesley, ISBN 0-201-51425-7.

Using this recursive definition of a list, it is possible to specify a recursive algorithm to process a list as follows:

```

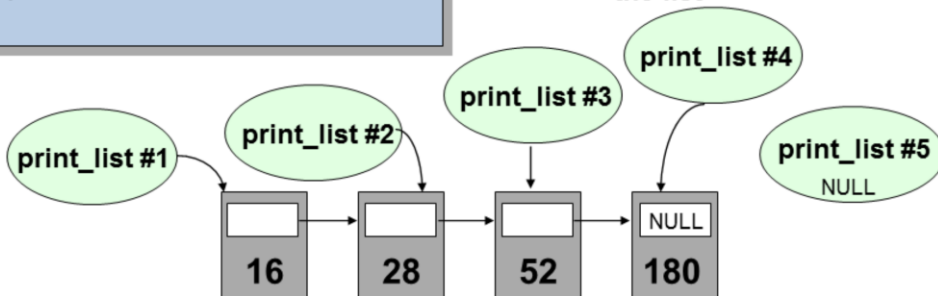
If the list is not empty
{
    Process the first item
    Then process the rest of the list (recursively!)
}
  
```

## Implementing a recursive print

- `print_list()` can be defined recursively as follows:

```
void print_list(node * p)
{
    if (p)
    {
        printf("%d\n", p->data);
        print_list(p->next);
    }
}
```

- ❶ Print current node
- ❷ Print remainder of the list



This example illustrates how graceful recursion can be. The `print_list()` function may be summarised as follows:

*If there are nodes*

*print the current node*

*print the remainder of the list.*

Many recursive functions have a similar appearance. The `if`-test at the beginning of the function ensures that the sequence of recursive function calls comes to an end when the list has been exhausted.

If the end of the list hasn't been reached, the current node is printed first. The `print_list()` then calls itself to repeat the print operation for the remaining nodes in the list.

When developing recursive algorithms, take care to ensure that the recursive function calls do come to an end. Consider the following incorrect recursive implementation of `print_list()`.

```
void print_list(node * p) /* Broken version */
{
    /* Recursive calls never end! */
    printf("%d\n", p->data);
    print_list(p->Next);
}
```

The chain of function calls never comes to an end. Each function call consumes an extra couple of bytes on the stack and the program will eventually crash because of stack overflow. When designing a recursive function you should always ensure that the depth of the recursive function calls does not exceed the available size of the stack.

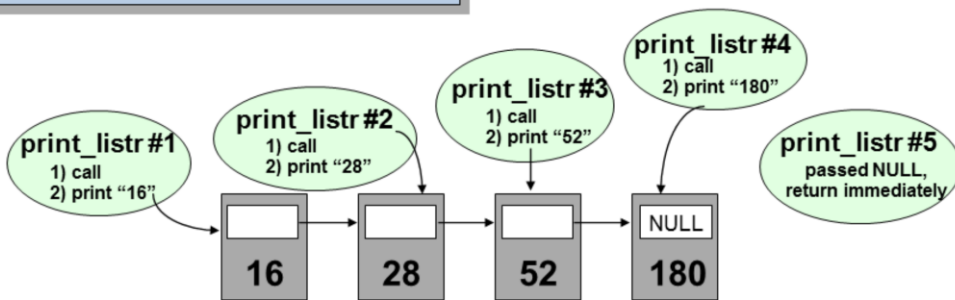


## Recursively printing backwards

- A recursive algorithm may be used to neatly print the list backwards

```
void print_listr(node * p)
{
    if (p)
    {
        print_listr(p->next);
        printf("%d\n", p->data);
    }
}
```

- ➔ ❶ Print remainder of the list *first*
- ➔ ❷ Then print the current node



This example shows a powerful feature of recursion: the ability to manipulate a linked list in reverse order. The trick is to invoke the function recursively *before* any individual manipulation of a node or item is performed.

Let us examine the sequence of events when `print_listr()` is first called.

Initially, `pNode` is pointing to the first node, which contains the integer value 16. The purpose of `print_listr()` is to print the rest of the list first, before printing the value 16. That is, the value 16 must be the last integer printed.

The function has a simple `if` test to tell us when we have passed the end of the list. `p` will be non-NULL the first time the function is called in the above example.

The value 16 can only be printed after the rest of the list has been printed.

Therefore, `print_listr()` is called recursively to print the rest of the list first. Taking a leap of faith, let us imagine that this works and does indeed print out the values 180, 52 and 28. When the recursive call to `print_listr()` returns, we can then print out the number 16. Therefore, the value 16 will be the last value that actually gets printed by this function.

We need to look a little more closely at how this mechanism works, and the easiest way to do so is to examine what happens when `pNode` is pointing at the last node (which contains the value 180 and the NULL value in `next`):

*The if condition is true, `print_listr` is called on `p->next` (`==NULL`) which returns. The data is printed (the 180), this invocation returns*

This condition happens at the end of the list. Note that this is the first occasion that `printf()` is executed; 180 is the first number to be displayed.

## Searching a linked list

- Searching for a particular value within a linked list must be done by a "brute force" approach

```
int search_list(node * p, int search)
{
    int result;
    while (p && p->data < search)
        p = p->next;
    if (p && p->data == search)
        result = 1; /* true */
    else
        result = 0; /* false */
    return result;
}
```

*Assumes data items  
are held in ascending  
order*

```
if (!search_list(head, 894))
    head = insert_list(head, 894);
```

- The list must be searched sequentially from the first item each time

Searching for a value within a linked list suffers from the same problems as searching through an array. When a large number of items are present it is a painfully slow process. There is no way to optimise the technique, other than to jump out of the loop when we find a value larger than the one we're searching for.

If you intend to do a lot of searching like this, you'd probably be better off with some form of tree. This is the subject of the next chapter.

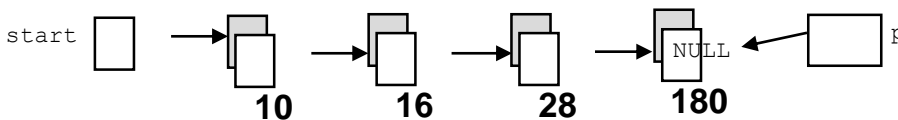
Although the algorithm uses iteration, recursive searching is also possible.

## Building a linked list

```
node * insert_list(node * p, int value)
{
    node * start = p;
    node * lag = 0;
    if (!p) {
        p = new_list_item(value);
        return p;
    }
    for (; p && p->data < value; p = p->next)
        lag = p;
    if (p && p->data == value) /* value already present */
        return start;          /* do nothing */
    if (!lag) {                /* insert before head */
        lag = new_list_item(value);
        lag->next = p;
        return lag;           /* new head */
    }
    lag->next = new_list_item(value); /* normal insertion */
    lag->next->next = p;
    return start;
}
```

```
node * head = 0;
head = insert_list(head, 16);
head = insert_list(head, 28);
head = insert_list(head, 52);
```

There are a number of things to consider when inserting items into a linked list. First is the problem that by the time we have found the insertion point, we have moved past it. For example, consider the insertion of 52 into the following list:



Working along the list, it is only when the 180 is being pointed to that we know the insertion point must be *before* the node containing 180. However, as previously discussed, since all the links point *forwards* there is no way to move back to the node containing 28.

For this reason, the pointer *lag* is declared. Its job is to lag one behind the current pointer. Thus when *p* points to the node containing 180, *lag* points to the node containing 28. It is then a straightforward matter of changing the next pointers to insert 52.

However, there are other cases to consider. For instance, what would happen if 2 was inserted into the list? Then *p* would point to the first node (containing 10) and *lag* wouldn't point anywhere. Then there is the case where there is no list at all. Then not only *lag* but *p* wouldn't point anywhere.

All these "special" cases make insertion into a linked list less straightforward than it might otherwise be.

Notice that the routine returns the address of the head of the list. This is very important since insertion of a value at the front will obviously cause the head to be changed. An alternative mechanism here would be to pass the address of the head of the list to *insert\_list*. Incidentally, the *new\_list\_item* function looks like:

```
node * new_list_item(int value)
{
    node * p = (node *)calloc(1, sizeof(*p));
    if (p == NULL)
        out_of_memory();
    p->data = value;
    return p;
}
```

## Deleting from a linked list

- Deleting a node in a linked list is more straightforward

```
node * delete_list(node * p, int value)
{
    node * start = p;
    node * lag = 0;
    for (; p && p->data < value; p = p->next)
        lag = p;
    if (!p || p->data != value)    /* didn't find it */
        return start;           /* nothing to remove */
    if (!lag)
        start = start->next;      /* deleting the head */
    else
        lag->next = p->next;      /* unsew this node */
    delete_node(p);
    return start;
}
```

Deleting a node from a linked list is more straightforward than insertion. There is the same point to consider in that once we have found the node to delete we need the address of the node before it. Thus the same `lag` pointer is used here.

Care must be taken when deleting the head of the list. This case is easily detected by the checking to see if the `lag` pointer is `NULL`.

The `delete_node` routine, in the case of a list which stored integers, would quite simply be a call to `free()`.

On the face of it, this is a quite useless routine. Why not replace the call to `delete_node` with a call to `free`? The main purpose of the routine is to provide a "hook" on which to hang things. Imagine the list contained not integers but words (arrays of characters which were dynamically allocated). Then freeing a node would not only involve deallocating the storage for the node itself, but also deallocating the dynamically allocated array of characters as follows:

```
void delete_node(node * p)
{
    free(p->data);
    free(p);
}
```

By already having the hooks into the `delete_node` routine in the code, changing the nature of the data contained by each node would be straightforward.

## Doubly linked lists

- **The elements of a linked list may have pointers pointing backwards as well as forwards**
- **Singly linked list**
  - One pointer per structure - high data/junk ratio
  - Difficult to traverse in "opposite" direction as already seen
- **Doubly linked list**
  - Two pointers per structure - lower data/junk ratio
  - Easy to traverse in both directions

---

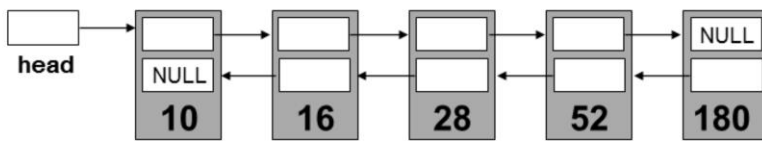
The one-way list has a very simple structure: there is a starting point with each node pointing to the next one and the last node pointing to `NULL`. This concept can be enhanced quite easily by giving each node the capability of pointing to its predecessor as well as its successor. This will increase the size of each node by one pointer, but achieves greater flexibility and creates a symmetry that makes the tasks of insertion and deletion easier and less error-prone. However, the implementation will require extra code in order to maintain the second pointer.

All data structures that can be implemented as one-way lists, such as stacks and queues, can be readily re-implemented using a two-way list.

## Representing nodes

- The representation is straightforward

```
struct double_list_node
{
    int                data; /* could be anything */
    struct double_list_node * next;
    struct double_list_node * prev;
};
typedef struct double_list_node dnode;
```



Representing a node to sit in a doubly linked list is very straightforward. Just add another pointer pointing backwards.

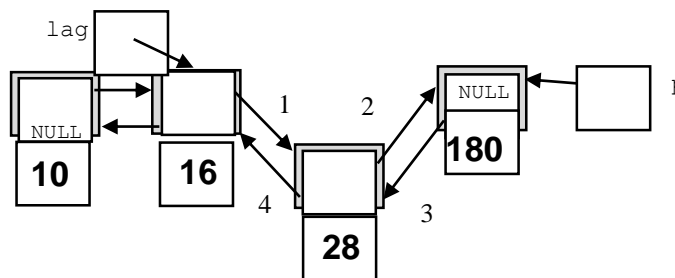
## Insertion into a doubly linked list

```

dnode * insert_double_list(dnode * p, int value)
{
    dnode * start = p, * lag = 0;
    if (!p) {                                /* list is empty */
        p = new_list_item(value);
        return p;
    }
    for (; p && p->data < value; p = p->next)
        lag = p;
    if (p && p->data == value)                /* value already present */
        return start;                        /* do nothing */
    if (!lag) {                              /* insert before head */
        lag = new_list_item(value);
        lag->next = p; lag->prev = 0; p->prev = lag;
        return lag;                          /* new head */
    }
    lag->next = new_list_item(value);        /* normal insertion */
    lag->next->next = p; p->prev = lag->next;
    lag->next->prev = lag;
    return start;
}

```

Consider the problem of inserting 28 into the following doubly linked list. By the time the `for` loop above has executed, the pointers `lag` and `p` will point to the nodes shown:



The assignment `lag->next = new_list_item(value)`

assigns the pointer labelled 1, creating the new node. In theory the `lag` pointer is not necessary since it would be possible to get to this node by using `p->prev`.

However if we were adding an item onto the end of the list (say 200) the pointer `p` would be `NULL` since we would have fallen off the end.

The next assignment `lag->next->next = p`

assigns the pointer labelled 2. This sets the new node's forward pointer to the node containing 180. Then the assignment `p->prev = lag->next`

sets the pointer labelled 3, making the node containing 180 point back to the new node. Finally the assignment `lag->next->prev = lag`

sets the pointer labelled 4, making the new node's previous pointer point to the node containing 16.

## Deletion from a doubly linked list

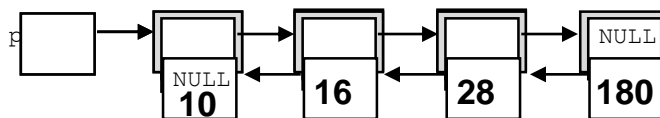
```

dnode * delete_list(dnode * p, int value)
{
    dnode * start = p, * lag = 0;
    for ( ; p && p->data < value; p = p->next)
        lag = p;
    if (!p || p->data != value)      /* didn't find it */
        return start;              /* nothing to remove */
    if (!lag) {                     /* deleting the head */
        start = start->next;
        if (start) start->prev = 0;
    }
    else if (!p->next) {            /* deleting the tail */
        lag->next = 0;
    } else {                       /* deleting a surrounded node */
        p->next->prev = lag; lag->next = p->next;
    }
    delete_node(p);
    return start;
}

```

When deleting a node in a doubly linked list there are three cases to consider:

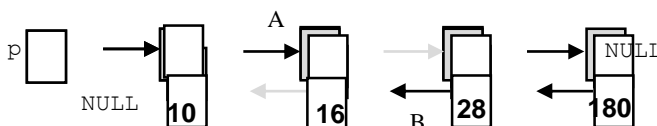
i) Deletion of the head of the list, e.g. delete node containing 10:



The only thing to do here is to make start point at the node containing 16, and to set the previous pointer for this node to be NULL. This makes the 16 node the head of the list.

ii) Deletion of the tail, e.g. delete the node containing 180. This is the most trivial since it merely requires that the next pointer in the 28 node be set to NULL. It then becomes the tail of the list.

iii) Deletion of a node surrounded by two other nodes, e.g. the node containing 16. There are only two pointers to worry about, those labelled A and B in the diagram below:

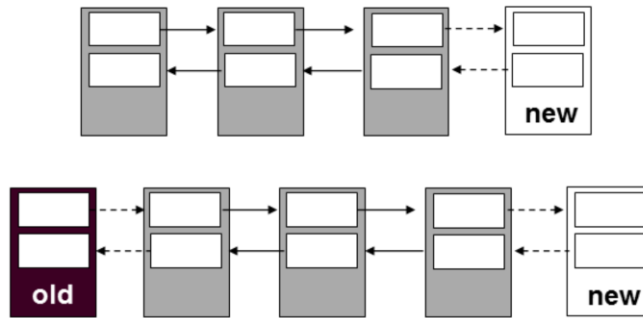


The pointers within the 16 node do not need to be reset since this node is about to be destroyed, thus it does not matter what they contain.



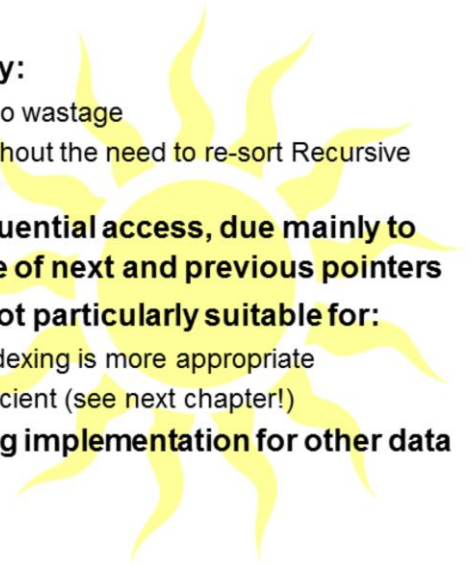
## Implementing other data structures

- **Once linked list code has been completed, other data structures may be implemented using the same code:**
  - Stacks where insertions and deletions are made at one end
  - Queues where insertions are made at the "front" and deletions at the "back"



Other dynamic data structures, like queues and stacks, are implemented using very similar constructs. Everything is achieved through the individual node's internal pointers. The data structures differ mainly in the rules which must be obeyed relating to addition, removal and traversing.

## Summary

- **The linked list is certainly a very commonly used dynamic data structure**
  - **As memory is allocated dynamically:**
    - There is efficient use of memory, i.e. no wastage
    - Items may be inserted and deleted without the need to re-sort Recursive techniques are often quite appropriate
  - **Lists are particularly useful for sequential access, due mainly to inherent ordering and the existence of next and previous pointers**
  - **The disadvantage is that lists are not particularly suitable for:**
    - Random access, where arrays and indexing is more appropriate
    - Sorted data, where trees are more efficient (see next chapter!)
  - **Lists may be used as the underlying implementation for other data structures like queues and stacks**
- 

The linked list is one of the most fundamental data structures. It turns out to be one of the most flexible. It provides the basis for simple one-way and two-way queues, stacks and rings. One of the most flexible and common two-way systems is the tree, which is the theme of the following chapter.

The flexibility is achieved mainly through the heavy use of dynamic memory. The number of items in a list is not known at compile time, so runtime memory allocation is a essential. Assuming that memory management is not a major problem, the programmer only has to worry about insertion and deletion of data nodes in order to maintain the dynamic nature of the data structure. Both of these processes, as well as actual data manipulation, can be performed using recursive algorithms if necessary. The list itself is a recursive structure, defined as a head (an item) and tail (the list of the remaining items). Care must be taken to measure the cost of implementing a recursive algorithm. Time and space overheads can be extensive.

Lists are superb for sequential access of dynamic data. As long as the maintenance and manipulation of the list has been implemented efficiently, data can be searched and processed relatively smoothly. However, lists are not so useful for random access or for holding and accessing sorted data. Arrays and trees, respectively, are better suited for these processes.