This chapter covers the notion of a linked list.

You must be comfortable with the use of a) structures and classes and b) pointers.
Primitives hold values

```java
int x = 10;
```

$x = 10$
Pointers hold addresses

```
int *x = 10;
```

Diagram: `x` pointing to the value `10`
Use `new` to allocate memory

```c++
int *iptr = new int(10);
// allocate an integer with initial value of 10

int *number = new int[10];
// allocate memory for 10 adjacent values (an array basically)

Customer *custptr = new Customer("Peppe", "LePew");
// allocate one customer object
```
Use `delete` to release memory

```cpp
int *iptr = new int(10);
delte iptr;

int *number = new int[10];
delete [] number;

Customer *custptr = new Customer("Peppe", "LePew");
delete custptr;
```
Dereferencing a pointer means getting a hold of its value

```cpp
int *iptr = new int(10);
cout << *iptr << endl; // displays 10

int *number = new int[10];
cout << number[0] << endl; // displays first element

Customer *custptr = new Customer("Peppe", "LePew");
cout << custptr->getFirstName(); // displays "Peppe"
```
int x = 10;
int *xptr = &x;
int *yptr = new int;
*yptr = x;

xptr is statically allocated on the stack, just like x and yptr.
yptr is assigned a dynamically allocated portion of memory, which is allocated on the heap.
Every time you use the "new" operator, you must pair it up with a "delete" operator. Thus, all memory allocated is returned.
What's the deal with arrays?

- Statically declared with a fixed size.
- Memory allocation is contiguous.
- Insertion / Deletion may require shifting.

If an array is statically declared, i.e. declared at compile time, then its size is fixed and potentially limiting. If the application needs more elements, the array would not be able to accommodate such needs.

You may use a dynamically allocated array however, and resize as the application needs change.

With either type of array, you still have the problem of memory allocation having to be continuous, a problem that can rear its ugly head with more demands for this resource.

Insertions and deletions in either array type can also cause a problem. If the array is to maintain order, then inserting may require the shifting of elements to free up an element to receive the new value. Similarly, deletion may require the shifting of elements to close the gap left by the removal of an element.
Dynamic arrays

- Size is automatically adjusted with insertions / deletions.
- Still must be shifted.
- Memory must be released.
Resizing: [1] Allocate

Original state:

```c
int* elements = NULL;
int numberOfElements = 0;
```

[1] Allocate

```c
newElements = new int[numberOfElements + 1];
```
Resizing: [2] Copy

// Note: the numberOfElements represents the size of the old array at this time.
for (int i = 0; i <= numberOfElements - 1; i++) {
    newElements[i] = elements[i];
} // If the old array contains elements all of them are copied.
// Note: the numberOfElements represents the size of the old array at this time.
newElements[numberOfElements++] = 1;

[3] Insert the new value
Resizing: [4] Release

// Note: All memory held by the old array is now returned to the heap
delete [] elements

[4] Release the old array

• newElements
  • 1 elements
Resizing: [5] Reassign

// Repeat steps 1-5 to add more elements
elements = newElements;
newElements = NULL;
How does a linked list stack up?

- Dynamically allocated with variable size
- Memory allocation can be non-contiguous
- Insertion /Deletion does not require shifting
So, what is a linked list?

A linked list is basically like a chain link. Each link is chained to the next link in the chain. This linking is accomplished with the use of pointers.

The data item could be as simple as a primitive type or a structured type (array, struct, class), or even a pointer to another linked list.

The head pointer refers to the beginning of the list and the optional tail if present to the end of the list. Using a tail makes inserting at the end a lot more efficient, since the entire list would not have to be traversed each time an insertion at the back is performed.
Creating a node with a struct

Node

D

struct Node

.item

.next
typedef <data type> ItemType;

struct Node {
    ItemType item;
    Node* next;

    Node(const ItemType& item, Node* const next)
        : item(item), next(next)
    {
    }
};

typedef – allows for the definition of a new type, based on an existing type. You are basically creating an alias that can abstract the actual type. This way there is less dependence on the actual data type used itself and creates a more generic node.

The const before the next parameter in the constructor guarantees that the parameter will not be allowed to change inside the function.

The use of the Node constructor makes it easier to initialize a Node object at declaration time. We will make use of this a lot in using linked lists.

Example: Node* head = new Node(item, next);

The nice thing about this constructor is that we can use it to allocate, populate and insert a node into a linked list. The next parameter will simply refer to one of the nodes in the linked list.
You can insert a node at three locations:

- **Front**

  - Head
  - Tail

- **Back**

  - Head
  - Tail

- **Within**

  - Head
  - Tail
Insert 1: At the front

Inserting at the front of the list is considered a special case, since the head pointer must be reassigned to the new node.

Since we created the Node constructor, we can create a new node whose next member points to head. Since head points to the first data node, the new node will link to this node via its next member.

Once head is assigned the new node the linked list is now complete. What used to be the first node is now the second, with head pointing to this new node.
Code to insert at the front

head = new Node(D, head);
Insert 2: At the back

Inserting at the back works in a similar fashion. If we are using a tail, then the tail will also have to be readjusted to point to the added node.

This is again demonstrated as two steps. The first creates the new node with its `next` member this time being `NULL`, to signify the end of the list.

The second steps completes the insertion by making two reassignments. The first is `tail->next`, which now must link to the new node, and second is `tail` which must be updated to refer to the new last node.
tail = tail->next
    = new Node(D,NULL);
This is the insertion that is performed more frequently on average. Here we introduce two new pointers, before and after, each one referring to the node preceding and succeeding the node to be added.

The final placement of the new node, and ultimately that of the before and after pointers, will be determined by the ADT being implemented. For instance, if we are implementing a position-oriented ADT, such as a List, then the ordinal position of the item being added will dictate where the node is added to the list. If on the other hand we are implementing a value-oriented ADT, such as a sorted List, then the value of the item being added will dictate.
Code to insert within

before->next = new Node(D,after);
We can see from the insertion examples so far that, when inserting at the front or the back of a list using a tail, the head and tail pointers will have to change.

When inserting within the list, we could use a before and after pointer to facilitate the insertion. Note that the after pointer is not absolutely necessary, since we could use before->next = new Node(D, before->next) to accomplish the same thing.
You can delete a node at three locations:

**Front**

- Head
- Tail

**Back**

- Head
- Tail

**Within**

- Head
- Tail
Here we introduce the `del` pointer, which points to the node to be removed.

After `head` has been readjusted, the node referred to by `del` is still present and must be deleted physically. In other words, its memory must be returned to the heap.

We will see how this is done shortly.
head = head->next;
When removing at the back, we must introduce the before pointer in addition to the del pointer. This pointer is needed, since the tail will have to change to point to the preceding node, which will become the last node after the removal of the currently last node.

Once more, note how the node pointed to by del is still here and must be removed.
before->next = del->next;
tail = before;
Delete 3: Within

One last time, note the del pointer and the node it points to that must be removed from the heap.

before->next = del->next;
before->next = del->next;
Delete observations so far

Delete front:
\[
\text{head} = \text{head} \rightarrow \text{next};
\]

both change

Delete back:
\[
\text{tail} = \text{before};
\]
\[
\text{before} \rightarrow \text{next} = \text{del} \rightarrow \text{next};
\]

Delete within:
\[
\text{before} \rightarrow \text{next} = \text{del} \rightarrow \text{next};
\]
Returning the deleted node to the heap

When deleting at the front...

...the back...

and within.

On all three examples, the node that was logically removed from the linked list is now physically removed by retiring its memory back to the heap.
del->next = NULL;  
delete del;  
del = NULL;

The first statement is simply a precautionary measure only. The idea behind it is as follows: If the node was returned to the heap without the next member set to NULL, then it would perhaps still point to a node on the list if and when it was reallocated by another new Node(...) operation in the future.

The second statement releases the memory back to the heap.

And the third is again a precautionary measure. Since we are done with the node, there is no reason to hold on to its address. This way, you will not accidentally try to dereference the pointer after the node it points to has been released.
Node* iter = head;

while (iter != NULL) {
    // Do something with the node
    iter = iter->next;
}

Traversing a linked list is similar to a sequential access of an array. You start at the first data node and move down the line, visiting each node in turn.

Visitation can mean any number of things, such as editing, displaying, or even removing.
Here the while condition has two components. The first `iter != NULL`, makes sure that the search will stop after checking the last node in the list. The second, `iter->item != keyValue` checks each node’s item against the value we are interested in finding.

At some point, either the value is found or the entire linked list is searched. You must check to see which of the two conditions was met, and that explains the `if` statement after the end of the loop.
Node *after = head, *before = NULL;

while (after != NULL && keyValue > after->item) {
    before = after;
    after = after->next;
}

// Did we find it?
if (after != NULL) {
    // Insert the node before the key node.
    // Insert at the front or back of the linked list, the if after the loop will have to be modified as follows.
    if (head == NULL) {
        // insert the very first node
    } else if (after == head) {
        // insert at the front
    } else if (after == NULL) {
        // insert at the back
    } else {
        // insert within
    }
}
Node *del= head, *before = NULL;

while (del != NULL && keyValue != del->item) {
    before = del;
    del = del->next;
}
// Did we find it?
if (del != NULL) {
    // Delete the key node.
}

To handle the removal of the first or last node, the // following the loop must be modified as follows:

if (del == head) {
    // delete the first node
} else if (del == tail) {
    // delete the last node
} else {
    // delete the key node
}
Implementing a List using a linked list
Code a List using a Linked List

// File: ListPointer.h

#ifndef LIST_POINTER_H
#define LIST_POINTER_H

#include <string>
using namespace std;

namespace LP {
    typedef string ListItemType;
}
// Class: ListException
class ListException {
private:
    string message;

public:
    ListException(const string& msg = "ListException thrown.")
        : message(msg) {} // end ListException()

    string what() const {
        return message;
    } // end What()
}; // end ListException

The first class is our exception class, ListPointer, modeled on the build-in exception class.
The constructor uses an optional parameter, so if the user of this class does not specify an error message, then the default message will be used.
The only method is the What() method that returns the actual error message itself.
Note that both methods are defined inline – the code itself is actually copied into the source code, rather than making a function call. This enhances performance a bit, but makes the executable's size a bit larger. Yet another trade-off.
Code a List using a Linked List

// Class: ListPointer
class ListPointer {
private:
  // Struct: Node
class Node {
public:
    ListItemType value;
    Node *next;
    Node(const ListItemType& value, Node * const next)
    : value(value), next(next)
    {} // end Node()
}; // end Node

The ListPointer class is defined next.
The Node structure is defined as a private member, conforming to the principle of information hiding, offered by encapsulation. No code outside the ListPointer class knows about this structure, thus avoiding unnecessary and restricting dependencies on implementation details.

Note that the Node constructor is also defined inline.
The utility functions added to our ListPointer class facilitate low level operations that are so often performed with linked lists. It is safe to say that these methods will appear in the majority, if not all, of your projects using linked lists.

The CreateNode() method is used to create and populate a new node. The method can also be used when inserting a node into the linked list. In that case the next pointer would point to a node in the linked list.

Finally, the NodeAt() method is used to locate the node containing the key value being searched. A pointer to this node is then returned.
The next set of methods are rather standard, with the `operator=` method justifying some further explanation.

This method is used whenever one object is assigned to another. The method performs a deep copy, thus avoiding the memory problems associated with a shallow copy, which is the default behavior of the assignment operator.

The method also performs a test (`this != &rhs`), to make sure that the two objects involved in the assignment are not actually one and the same. If that was the case, then the program would crash, since the `DeleteNodes()` call would effectively wipe out the linked list the `CopyNodes()` would try to duplicate.
Code a List using a Linked List

// Returns the numberOfNodes.
int count() const;

// Inserts the item at the specified pos. Throws a
// ListException if pos is out of range, or node allocation
// fails.
void insertAt(const int pos, const ListItemType& item);

// Removes the node at pos. Throws a ListException if pos is
// out of range.
void removeAt(const int pos);

// Retrieves the node at pos. Throws a ListException if pos
// is out of range.
const ListItemType& retrieveAt(const int pos) const;
Code a List using a Linked List

// Searches for the item on the list. If the item is found,
// atPos holds its position and method returns true. If the
// item is not found, atPos is undefined and method returns
// false.
bool contains(const ListItemType& item, int& atPos) const;

// Returns true if list is empty, false otherwise.
bool isEmpty() const;
}; // end ListPointer
} // end namespace

#endif
The `DeleteNodes()` function deletes each node in succession, starting with the first data node. The removal is done in two steps: first, the logical removal with `head = head->next`, and then the physical removal with the next two lines inside the while loop.
The `copyNodes()` makes an exact duplicate of the parameter object. The copy is performed in two distinct steps.

First, you create the first data node which requires the initialization of both the `head` and the `tail` pointers.

Second, the remaining nodes are created by adding each node to the end of list.

If the allocation fails for any reason a `bad_alloc` exception is raised and handled within this method. The actual error message is displayed on the screen.
Code a List using a Linked List

// Allocates and populates a node. Throws a bad_alloc exception
// if node allocation fails.
ListPointer::Node*
ListPointer::createNode(const ListItemType&
    value, Node * const next) {

    // Throw bad_alloc exception if unable to allocate memory.
    Node *nptr = new Node(value, next);

    if (nptr == nullptr) throw bad_alloc();

    return nptr;
} // end createNode()
Code a List using a Linked List

// Returns a pointer to the node at pos. Throws a ListException
// if pos is out of range.
ListPointer::Node* ListPointer::nodeAt(const int pos) const {
    Node *iter;
    if (pos < 0 || pos > numberOfNodes) {
        throw ListException("nodeAt() error: pos out of range");
    } // end if
    if (pos == 0) {
        iter = NULL;
    } // end if
    else {
        iter = head;
        for (int skip = 1; skip < pos; skip++) {
            iter = iter->next;
        } // end for
    } // end else
    return iter;
} // end nodeAt()
// Sets head to NULL and numberOfNodes to 0.
ListPointer::ListPointer() {
    head = tail = nullptr;
    numberOfNodes = 0;
} // end ListPointer()

// Calls the CopyNodes() method.
ListPointer::ListPointer(const ListPointer& o) {
    copyNodes(o);
} // end ListPointer()

// Calls the DeleteNodes() method.
ListPointer::~ListPointer() {
    deleteNodes();
} // end ~ListPointer()
As mentioned before, the `operator =()` method checks for self-assignment before deleting the current object and recreating it.
// Inserts the item at the specified pos. Throws a ListException
// if pos is out of range, or node allocation fails.
void ListPointer::insertAt(const int pos, const ListItemType& item)
{
    // Check pos and if out of range, throw exception.
    if (pos < 1 || pos > numberOfNodes + 1) {
        throw ListException("Insert error: pos is out of range.");
    } // end if
    Node *before = nodeAt(pos - 1);

    // What happens if we are inserting at the front?
    try {
        if (before == nullptr) {
            head = createNode(item, head);
        } // end if
        else {
            Node *after = before->next;
            before->next = createNode(item, after);
        } // end else
    } // end try

Note how inserting at the front of the list is handled. If the value of before is NULL, then it means we are inserting at the front of the list. This includes inserting the very first node as well as subsequent first nodes.

The else portion handles inserting within the list as well as at the back. Make sure you verify this by drawing the appropriate diagram.

The method throws two type of exception: First a ListException if pos is out of range and a bad_alloc if memory allocation fails.
Code a List using a Linked List

```cpp
catch (bad_alloc ba) {
    throw ListException("Insert error: Unable to allocate needed
                     memory.");
} // end catch
numberOfNodes++;
}// end insertAt()

// Removes the node at pos. Throws a ListException if pos is
// out of range.
void ListPointer::removeAt(const int pos) {
    // Throw exception if pos out of range.
    if (pos < 1 || pos > numberOfNodes) {
        throw ListException("Remove error: pos is out of range.");
    }

    Node* before = nodeAt(pos - 1);
    Node* del;
```

56
Code a List using a Linked List

// Handle removing the first node.
if (before == nullptr) {
    del = head;
    head = head->next;
} // end if
else {
    del = before->next;
    Node *after = del->next;
    before->next = after;
} // end else

del->next = nullptr;
delete del;
before = after = del = nullptr;
numberOfNodes--;
} // end removeAt()
// Retrieves the node at pos. Throws a ListException if pos
// is out of range.
const ListItemType& ListPointer::retrieveAt(const int pos) const {
    // Throw exception if pos out of range.
    if (pos < 1 || pos > numberOfNodes) {
        throw ListException("Retrieve error: pos is out of range.");
    } // end if
    Node *iter = nodeAt(pos);
    return iter->value;
} // end retrieveAt()

// Returns true if list is empty, false otherwise.
bool ListPointer::isEmpty() const {
    return bool(head == nullptr);
} // end isEmpty()
The `Contains()` method searches for a key node and if it is found its position within the linked list is recorded in `atPos` and the method returns true. If the key node is not found, then `atPos` is undefined and the method returns false.
if (iter == nullptr) {
    found = false;
} // end if

return found;
} // end contains()

} // end namespace LP
Code a List using a Linked List

File: ListPointerDriver.cpp

#include "ListPointer.h"
#include <iostream>

using namespace std;
using namespace LP;

int main() {
    ListPointer list;
    int pos;
    cout << "Uninitialized list: Length = " << list.count()
        << endl;

    // The driver instantiates a ListPointer object list using the default constructor.
    // This creates an empty list, as verified by the screenshot.
// Add a few values to the list.
list.insertAt(1, "A");
list.insertAt(2, "B");
list.insertAt(1, "C");

cout << endl
  << "Initialized list: Length = " << list.count()
  << endl;

for (int i = 1; i <= list.count(); i++) {
  cout << list.retrieveAt(i) << endl;
} // end for

Next three items are added to the list and displayed one per line.
The list is then searched for the presence of the item "A" which is found at pos 2 on the list.
Code a List using a Linked List

// Testing the copy constructor.
ListPointer list2(list);
list2.insertAt(4, "D");

cout << endl << "list2: ";
for (int i = 1; i <= list2.count(); i++) {
    cout << list2.retrieveAt(i) << " ";
} // end for

cout << endl << " list: ";
for (int i = 1; i <= list.count(); i++) {
    cout << list.retrieveAt(i) << " ";
} // end for

Next the copy constructor is tested by creating a second list, list2, which is an exact duplicate of the first list. A new item is added to the second list and then both lists are displayed.

As the screenshot verifies, the original list has not changed by the addition of the new item into the second list. This verifies that the lists are different and that the copy constructor has indeed created a duplicate.

If the two list displays were the same, that would have indicated a shallow copy and an improper implementation of the copy constructor.
// Testing the overloaded operator =().
ListPointer list3;
list3 = list;

list3.insertAt(1, "S");

cout << endl << endl << "list3: ";
for (int i = 1; i <= list3.count(); i++) {
    cout << list3.retrieveAt(i) << " ";
} // end for

cout << endl << " list: ";
for (int i = 1; i <= list.count(); i++) {
    cout << list.retrieveAt(i) << " ";
} // end for

cout << endl << endl;
return 0;
} // end main()

The same test is now performed with the overloaded operator =(). As the screenshot verifies, the operator method has indeed created a duplicate.
Code a List using a Linked List

Uninitialized list: Length = 0
Initialized list: Length = 3
List does contain the item 'A' at pos = 2
list2: C A B D
list: C A B
list3: S C A B
list: C A B
Press any key to continue . . .
Eliminate front end complications by using a head node

Without a head (dummy) node:

With a dummy node:
Inserting at the front does not change head

new Node(D, head->next);

head->next = new Node(D, head->next);
head->next = new Node(D, head->next);
Deleting at the front does not change head

```
head->next = del->next;
```
Code to delete at the front
<head node linked list>

head->next = del->next;
Always start with the first data node when finding or traversing

iter = head->next;
The `NodeAt()` method is simplified with the elimination of the if else statement.

The assignment of `iter = head` will return the head node if the `pos` is 0, as this would be the node before the one being inserted or deleted.
Take note of the head node implementation changes

```cpp
// ListPointer
ListPointer::ListPointer() {
    head = NULL;
    numberOfNodes = 0;
} // end ListPointer()

// After
ListPointer::ListPointer() {
    head = CreateNode(ListItemType(), NULL);
    numberOfNodes = 0;
} // end ListPointer()
```

The constructor must create the dummy node. Notice the use of ListItemType() to initialize the item to the default value of the underlying data type. This default value is basically 0 for numeric types, false for bool types, "" for strings, and 0 for chars.
The insert is also simplified with the elimination of the if else statement.
The iter pointer will either point to the dummy node for insertions at the front, or some other node if inserting within or at the back.
Take note of the head node implementation changes

```c
// RemoveAt
if (iter == NULL) {
    del = head;
    head = head->next;
} else {
    del = iter->next;
    iter->next = del->next;
}// end else

// After
del = iter->next;
iter->next = del->next;
```

The removal is simplified as well, since the `iter` pointer will be either point to the dummy node or some other node in the list.
Take note of the head node implementation changes

// IsEmpty
bool ListPointer::IsEmpty() const {
    return bool (head == NULL);
} // end IsEmpty()

// After
bool ListPointer::IsEmpty() const {
    return bool (head->next == NULL);
} // end IsEmpty()

Here note that an empty linked list contains only the dummy node.
A circular linked list can be very handy when you need to keep cycling through the nodes. The last node is made to point to the first node in order to complete the cycle.

With a circular linked list we drop the head pointer and rename the tail to list. Only one pointer is needed to refer to the last node (tail) and the first node (tail->next).
Insert a node at the back of a circular linked list

Inserting at the back requires the adjustment of the list pointer, which in made to point to the new node. The new node is now the last node in the linked list.
Code to insert at the back
<circular linked list>

```c
list = list->next = new Node(D, list->next);
```
Insert a node at the front of a circular linked list

Insert a node at the front of a circular linked list:

```
new Node(D, list->next);
```

Inserting at the front does not require the adjustment of list in any way.

```
list->next = new Node(D, list->next);
```
Code to insert at the front
<circular linked list>

list->next = new Node(D, list->next);
Delete a node at the back of a circular linked list

Deleting the last node again requires the adjustment of the list pointer. The pointer `del` is used to point to the node that is being deleted.

Note that the deleted node must still be removed from the heap. That code is not shown here.
before->next = del->next;
list = before;
Delete a node at the front of a circular linked list

Delete the first node does not require the list to change. Again, the deleted node must be removed from the heap. That code is again not shown here.
Code to delete at the front
<circular linked list>

```c
before->next = del->next;
```
Code to traverse a linked list
<circular linked list>

Node* first = list->next;
Node* iter = first;

do {
    // process the visited node.
    iter = iter->next;
} while(iter != first);
A doubly linked list allows for a bidirectional navigation of the linked list.
To turn a single node into a double node, just add an additional pointer, perhaps called previous, that points to the node before it.
We added the `previous` pointer as a member of the node structure to facilitate backward navigation.

The constructor is also updated to reflect the addition of the new member.
When using a doubly linked list, each end of the list can require the adjustment of the head and optionally of the tail pointers.

For that reason we employ two dummy nodes, one at each end, to make the need for this adjustment unnecessary.

Note that in this case, an empty linked list contains no data nodes, but two dummy nodes.
Insert 1: At the front

new Node(D, head, after);

head->next = after->previous = new Node(D, head, after);

Inserting at the front does not require head to be adjusted, similar to the singly linked list.
head->next = after->previous
            = new Node(D, head, after);
Similarly when inserting at the back, the tail does not change either.
Code to insert at the back
<doubly linked list>

before->next = tail->previous
= new Node(D, before, tail);
Insert 3: Within

```
new Node(D, before, after);
```

```
before->next = after->previous = new Node(D, before, after);
```
Code to insert within a doubly linked list:

```java
before->next = after->previous
            = new Node(D, before, after);
```
Delete 1: At the front

Deleting at the front when using a dummy node by now should be easy to follow.

Again observe that the deleted node must be removed from the heap. This code is not shown here.
Code to delete at the front
<doubly linked list>

head->next = after;
after->previous = head;
Delete 2: At the back

before->next = tail;
tail->previous = before;

Same explanation here as in the previous case of deleting at the front.
Code to delete at the back
<doubly linked list>

before->next = tail;
tail->previous = before;
Delete 3: Within

before->next = after;
after->previous = before;
Code to delete within <doubly linked list>

before->next = after;
after->previous = before;
Similar to the case of a single node, we clear the two pointer members and then release the node.

del->next = NULL;
del->previous = NULL;
delete del;
del = NULL;
Consider the following when choosing between an array and a linked list:

Ease of coding:

An array is definitely easier to code since it is rather straightforward, and you don't have to manage all the additional pointers.

There are a lot of well-known algorithms for dealing with arrays and this makes their use easier for most people.
Consider the following when choosing between an array and a linked list:

- **Memory usage**

An array requires memory for the elements and that is all. A linked list requires some additional memory for the use of the linking pointers, and thus will consume a bit more memory.

Note however that if the size of the item is substantially larger than 4 bytes (the size of the pointer), then adding 4 more extra bytes does not have a significant impact, especially if the linked list contains a large number of nodes.
Consider the following when choosing between an array and a linked list.

Insertions / Deletions

Insertions and deletions from an array may require shifting of elements, an operation that is quite time consuming, particularly when repeated multiple times. Note however that there are times where you can limit the insertions/deletions to the end of the array, and this can avoid the shifting.

A linked list never incurs the shifting penalty, as all there is to the insertion/deletion is a shifting of pointers. Due to this efficiency, linked lists are better suited over arrays when shifting would otherwise occur.

Static arrays suffer from a fixed size, a problem that can be overcome with a dynamic array, however the shifting can still be present even in dynamic arrays.

Linked lists have no upper limit restriction and can grow as the data dictates.
Sequential navigation of each data item is comparable for both an array and a linked list. One data structure does not offer any advantage over the other, since all items must be visited in turn.
Consider the following when choosing between an array and a linked list.

When it comes to accessing individual items at random the array has the advantage.

The time to access the first time is the same as the time to access the second, third and last. This access time is constant for each element of the array.

A linked list has to be traversed each time a node is accessed, thus the time to access the first node will be less than the time to access the last node. The time is linear, or proportional to the number of nodes.
Consider the following when choosing between an array and a linked list.

Searching for a specific value when the data structures cannot be sorted first is a tie, since both will use a linear search approach.

If the data is sorted on the other hand, the array offers the advantage of the binary search, something the linked list cannot use effectively.
Consider the following when choosing between an array and a linked list:

Sorting is another operation that the array has the advantage, since there are a lot of algorithms to perform such sorting.

Linked lists are prohibitively hard to sort and thus don't offer the same advantages as the arrays do.
Concluding Remarks

- No one implementation will be 100% optimum against all required operations.
- Make the common operations fast.
- Take the hit for the less common.
- It is always a trade-off.