Linked Lists

You should have seen linked lists already from your previous programming course, but we'll still give a quick overview and then talk about variations of linked lists.

Arrays are useful in many applications but suffer from two significant limitations:

1. The size of the array must be known at the time the code is compiled
2. The elements of the array are the same distance apart in memory, requiring potentially extensive shifting when inserting a new element

This can be overcome by using linked lists, collections of independent memory locations (nodes) that store data and links to other nodes. Moving between the nodes is accomplished by following the links, which are the addresses of the nodes. There are numerous ways to implement linked lists, but the most common utilizes pointers, providing great flexibility.

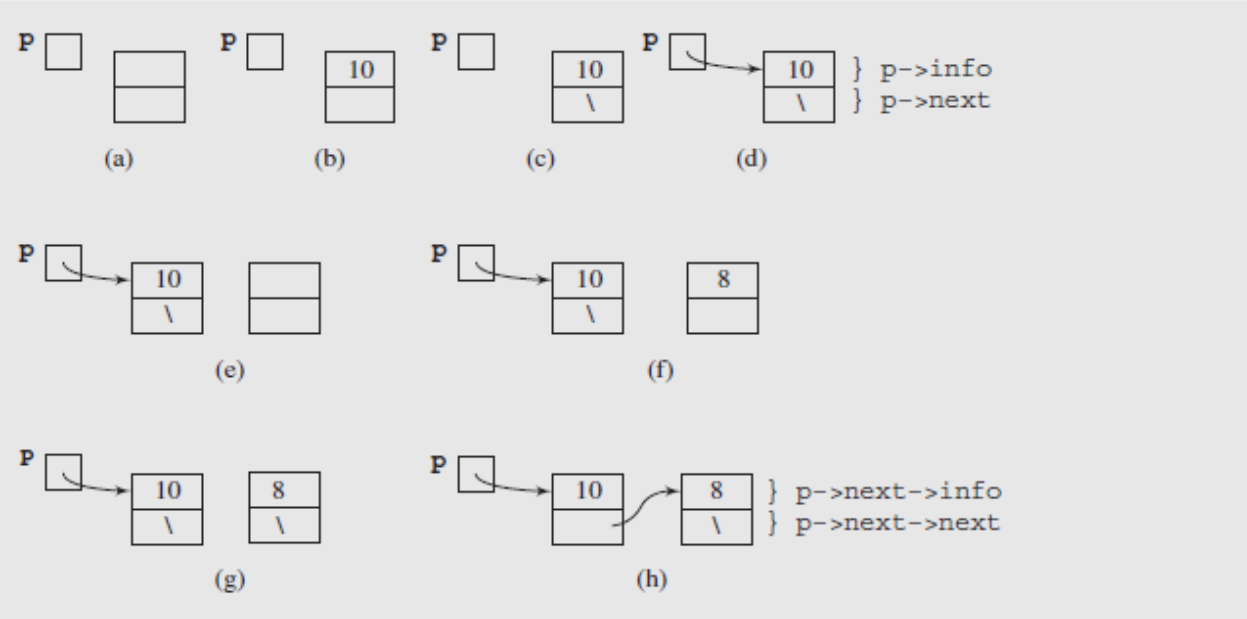
The simplest type of linked list is a **singly linked list**. We have a "node" which is generally a class or a struct. The node contains data and a pointer to the next node in the list. The last node in the list has a null pointer.

Here is a simple node definition:

```
class IntSLLNode
{
public:
   IntSLLNode();
    IntSLLNode(int i, IntSLLNode *in);
   int info;
   IntSLLNode *next;
};

IntSLLNode::IntSLLNode() {
      next = nullptr;
}
IntSLLNode::IntSLLNode(int i, IntSLLNode *in = nullptr) {
      info = i;
      next = in;
}
```

Here is a simple list we could build with:

```
IntSLLNode *p = new IntSLLNode(10);

p->next = new IntSLLNode(8);
```



If you are compiling on Linux, just a reminder that you may need to use the g++ flag -std=c++11

Let's cover some common operations on linked lists:

- Insertion
- Deletion
- Search

Insertion

We generally have three places we can insert a node into the list: The front, the end, and the middle. Inserting in the front requires us to do the following:

1. temp = new Node
2. temp->info = value
3. temp->next = head
4. head = temp

Inserting a node at the end of the list requires we have a pointer to the last node in the list. We can either maintain this ourselves or loop through the list from the head to the end to get the tail.

1. temp = new Node
2. temp->info = value
3. temp->next = null
4. tail->next = temp
5. tail = temp

Note that if the list is empty then we will fail on step 4, so we typically have to make a special check for empty lists to handle adding or removing elements.

Inserting into the middle of the list requires a pointer to the node we want to insert after. If p points to the node we want to insert the new data after:

1. temp = new Node
2. temp->info = value
3. temp->next = p->next
4. p->next = temp

Deletion

First, let's look at deleting from the head of the list. If the list is empty, then there is nothing to do. You should make sure to check for this condition or you can get errors accessing null pointers! Otherwise we simply move to the next item:

```
temp = head;

head = head->next;

delete temp;
```

Deleting from the middle of the list requires a pointer to the node BEFORE the node you want to delete. Let's say p is the pointer to the "previous" node before the one we want to delete. Then:

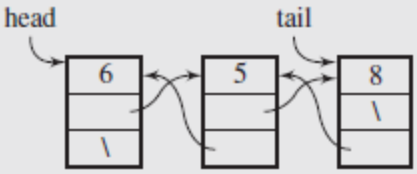
```
temp = p->next;

p->next = temp->next;

delete temp;
```

Search

Searching for an item in the list requires simply iterating through the list until reaching the desired item, or null. The expected search time is O(n).

Doubly-Linked Lists

A doubly-linked list maintains pointers in both directions. For example, with a singly linked list, if we want to delete the tail, we need to iterate through the list to get to the node pointing to the tail. To address this problem, and others such as deleting from the middle of the list, we can add a second pointer to each node that points to the previous node.



We now have one more variable in our class. Here is a version that uses templates so we can make a doubly-linked list of any type. In this example we only implement two doubly-linked list functions, adding to the tail, and deleting from the tail:

```
#include <iostream>
#include <string>

using namespace std;

template<class T>
class DLLNode
{
public:
	DLLNode();
	DLLNode(const T &el, DLLNode *n, DLLNode *p);
	T info;
	DLLNode *next, *prev;
};

template<class T>
DLLNode<T>::DLLNode()
{
	next = prev = nullptr;
}

template<class T>
DLLNode<T>::DLLNode(const T &el, DLLNode *n, DLLNode *p) :
	info(el), next(n), prev(p)
{
}

template<class T>
class DoublyLinkedList
{
public:
	DoublyLinkedList();
	void addToDLLTail(const T &el);
	bool deleteFromDLLTail(T &el);
protected:
	DLLNode<T> *head, *tail;
};

template<class T>
DoublyLinkedList<T>::DoublyLinkedList() : head(nullptr), tail(nullptr)
{
}
```

```
template<class T>
void DoublyLinkedList<T>::addToDLLTail(const T &el)
{
	if (tail != nullptr)
	{
		tail = new DLLNode<T>(el, nullptr, tail);
		tail->prev->next = tail;
	}
	else
		head = tail = new DLLNode<T>(el,nullptr,nullptr);
}

template<class T>
bool DoublyLinkedList<T>::deleteFromDLLTail(T &el)
{
	if (tail != nullptr)
	{
		el = tail->info;
		if (head == tail)
		{
			delete head;
			head = tail = nullptr;
		}
		else
		{
			tail = tail->prev;
			delete tail->next;
			tail->next = nullptr;
		}
		return true;
	}
	return false;
}

int main()
{
	DoublyLinkedList<string>  dll;
	dll.addToDLLTail("hello");
	dll.addToDLLTail("there");
	dll.addToDLLTail("A");
	dll.addToDLLTail("B");
	string element;
	while (dll.deleteFromDLLTail(element))
		cout << element << endl;
	system("pause");
}
```

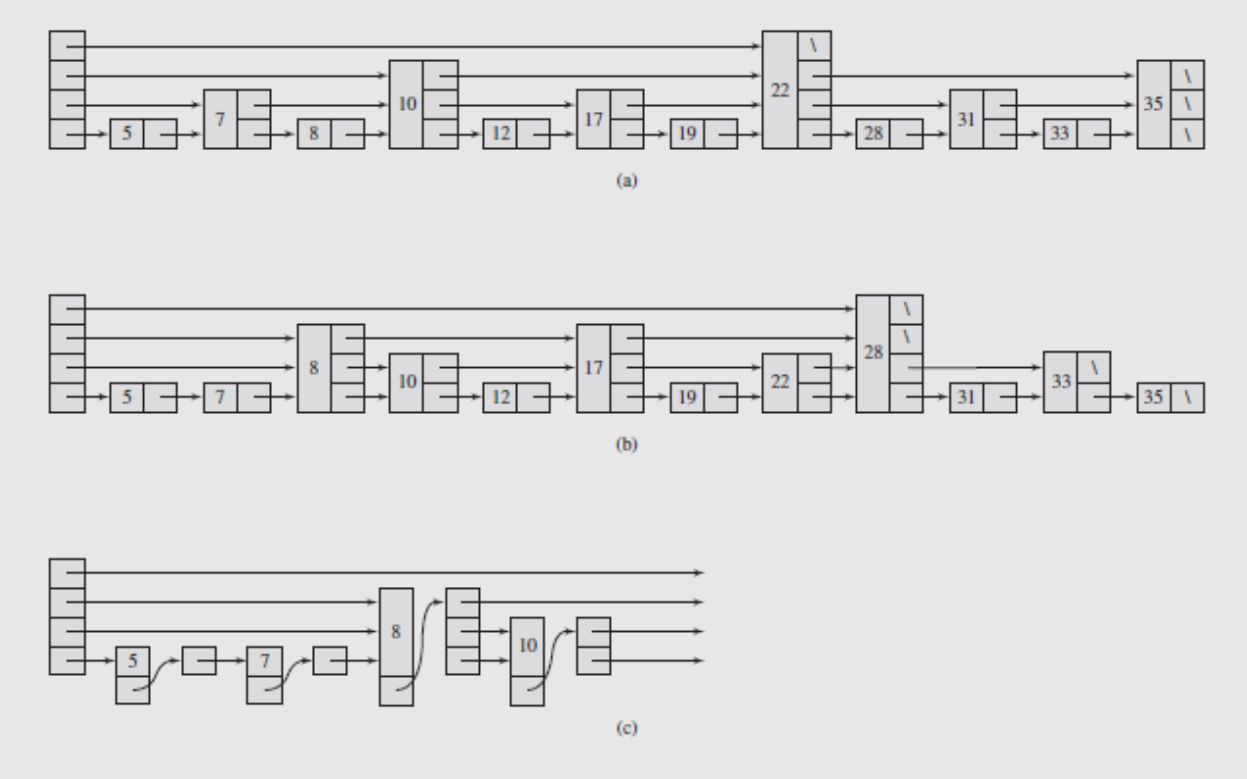
Note my version is a little different from the book. I use the more modern “nullptr” as well as classes instead of structs and separate out some of the implementation from the class definition.

We’ve left out cases like deleting from the middle or inserting into the middle – these will be coming for you as exercises ☺

Skip Lists

The biggest drawback to the linked lists we've examined is that they are sequential in nature. Searching for a particular element requires we look at every node in the list starting from head until we find the target or reach the end of the list. While ordering the elements on frequency of access can help, the processing is still sequential. To improve on this, a variation called a ***skip list*** can be used to implement non-sequential searching of a linked list. We require the list sort the elements in ascending order.

The idea is that every second node points to the node two positions ahead and every fourth node points to the node four positions ahead, and so on.



A skip list with (a) evenly and (b) unevenly spaced nodes of different levels; (c) the skip list with pointer nodes clearly shown

- In general, in a list of *n* nodes, for *k* such that $1 \leq k \leq \lfloor \lg n \rfloor$ and *i* such that $1 \leq i \leq \lfloor n/2^{k-1} \rfloor$, the node at location $2^{k-1} * i$ points to the node at location $2^{k-1} * (i - 1)$
- This is implemented by having different numbers of pointers in the nodes – half have one pointer, a quarter have two pointers, and so on
- The number of pointers determines the ***level*** of the node, and the number of levels is $\lfloor \lg n \rfloor + 1$

Search idea:

- To find a target value in the list, we start with the highest level pointers

- These are followed until we find the target, reach the end of the list, or reach a node whose data value is larger than the target
- In the latter two cases, we back up to the node preceding the larger data value, go to the next level of pointers, and restart
- This process is repeated until we find the target, or reach the end of the list or a larger value when traversing the first-level links

Pseudocode for search:

```
Search(element el)
    p = the nonempty list on the highest level i
    while el not found and i != 0
        if p->key > el
            p = a sublist that begins in the predecessor of p on level i-1
        else if p->key < el
            if p is the last node on level i
                p = a nonempty list that begins in p on the highest level < i
                i = the number of the new level
            else
                p=p->next
```

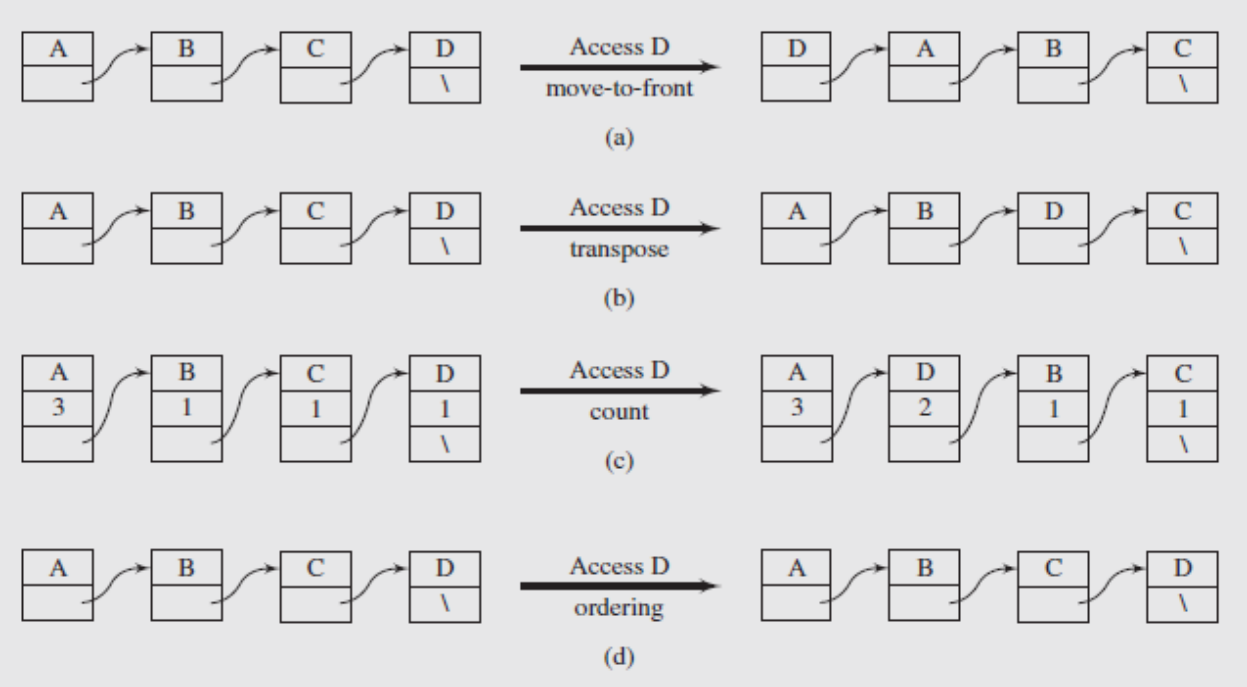
Searching is now more efficient, at the expense of insertion and deletion operations. If we have lgn levels then the search time is O(lgn). In the worst case, all the lists are on the same level, and we get a regular linked list. Inserting a new node, for example, requires restructuring of the nodes that follow the new node. In particular, the number of pointers and their values must be changed.

Self-Organizing Lists

Another way to speed up search in a list is to dynamically re-organize the list as it is used. There are several ways to do this. Here are four approaches:

1. Move to front. When the target is found, it is moved to the front of the list
2. Transpose. When the target is found, it is swapped with its predecessor in the list
3. Count. The list is ordered by frequency of access
4. Ordering. The list is ordered based on the natural nature of the data (e.g. alphabetical, known types such as prioritizing dogs over birds if dogs are more popular)

Runtime for 1 and 2 is O(1) for a single move or transpose. Count might take O(n) to scan back through the list to find the correct location for the newly increment count.



We can insert new nodes either in the front or at the end. How do we analyze the runtime of these algorithms? The efficiency is going to vary based upon the type of data access.

A typical approach that is used is to measure the runtime experimentally. Come up with an actual dataset and measure how many operations it takes to search a number of elements, perhaps randomly selected or artificially selected.

Another approach is to compare to the **optimal static ordering**. We order the data by the frequency of their occurrence in the search list. This only works for a static search test set. Using this as a comparison, the count and move-to-front methods are, in the long run, at most twice as costly as the optimal.

There is general improvement for all techniques as the size of the data increases. For small lists, a simple linked list suffices.

The Standard Template Library in C++ includes a list template type that is implemented as a doubly-linked list to the head and the tail. The following code demonstrates common list operations; see a reference for a listing of other functions.

```
#include <iostream>
#include <string>
#include <list>

using namespace std;

void printList(list<string> lst)
{
	for (string s : lst)
	{
		cout << s << " ";
	}
	cout << endl;
}

int main()
{
	list<string> lst;

	lst.push_back("A");
	lst.push_back("B");
	lst.push_back("C");
	printList(lst);  // A B C

	lst.push_front("First");
	printList(lst); // First A B C

	lst.remove("B");  // First A C ; removes all nodes with B
	printList(lst);

	auto iterator = lst.begin();  // iterator or "pointer" to first element
	cout << *iterator << endl;  // First
	iterator++;
	cout << *iterator << endl;  // A

	// Erase the element referenced by the iterator
	lst.erase(iterator);
	printList(lst);   // First C

	// Erase everything
	lst.erase(lst.begin(), lst.end());
	printList(lst);  // Nothing

	lst.push_back("D");
	lst.push_back("C");
	lst.push_back("B");
	lst.push_back("A");   // D C B A
	cout << lst.size() << endl;    // 4
	string s = lst.front();
	cout << s << endl;   // D
	lst.pop_front();
	printList(lst); // C B A
```

```
        iterator = lst.begin();
        iterator++;  // Points to B
        lst.insert(iterator, "Z");  // C Z B A
        printList(lst);

        lst.sort();
        printList(lst); // A B C Z;  requires overloaded comparison for type T

        system("pause");
}
```