COSC2430— Spring 2021

Lists and iterators

1. Singly linked lists

The UBVector class and dynamic arrays (http://en.wikipedia.org/wiki/Dynamic_array) in general have several disadvantages:

- Inserting and removing an element somewhere in the middle of the sequence takes linear time. urthermore, if the items we hold in the sequence are large, the cost of shifting a suffix of the array own one position is proportionally large.
- A vector wastes $\Omega(n)$ space in the worst case, where the constant inside $\Omega(n)$ is propotional to the item size. We have to allocate space to hold about n items.
- A vector is stored in a contiguous block of memory in the freestore (there's a subtle difference (h p://www.gotw.ca/gotw/009.htm) with the heap but we couln't care less). Sometimes, it is possible that the freestore has sufficient free space to store all elements of a vector, but it doesn't have a sufficiently large contiguous block of memory to store the vector. This is called the fragmentation problem (h p://en.wikipedia.org/wiki/Fragmentation_(computing)), which is very common in all kinds of memory/disk allocation tasks in operating system design.
- Inserting and deleting from a vector are highly global operations: we shift an entire suffix of the underlying array. List updates are highly local: we only need to lock two adjacent elements. This advantage cannot be overlooked in a concurrent environment because we may want multiple threats to modify/access a llong list at the same time. In fact, locality is also a key advantage (h p://drdobbs.com/article/print?articleId=208801371&siteSectionName=parallel) of linked lists over more elaborate search structures such as various types of binary search trees.

Linked list (http://en.wikipedia.org/wiki/Linked_list) is a classic container (http://www.cplusplus.com/reference/stl/) data structure which overcomes those drawbacks. A singly linked list is a sequence of "nodes" chained together by pointers: one node has a pointer to the next node. The last node typically points to `NULL` which is a pseudonym for `0`. We can't store things at this `NULL` address. Each node stores a "payload" which could be of any type, just as an item in a vector can be of any type. Linked lists are the core data structure in the LISP programming language (http://en.wikipedia.org/wiki/Lisp_(programming_language)). (LISP stands for, well, *LISt Processing*.")

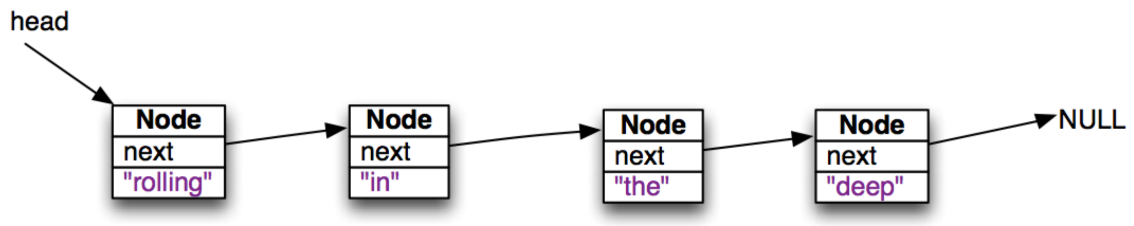
Herbert Simon won the Turing award in 1975 partly due to the invention of linked lists. He also predicted in the 60s that a computer will play chess better than the best human within 10 years. His prediction was off by about 20 years, but his linked list lives much longer than that.

While a vector allows for random access to elements in the vector (using the subscripting operator, indexing elements is efficient), a linked list does not have that capability. If we know the address of the top element of a vector, we can compute the address of (and thus access) the kth element in a vector easily using pointer arithmetic. This is possible because all elements of a vector are stored contiguously in memory. Accessing a linked list is intrinsically a sequential operation. In a linked list, elements do not have to be stored contiguously. Addresses of elements are basically arbitrary locations in the freestore. To get to a particular element, we typically have to traverse the list down the pointer chain until we meet the wanted element.

The following example should clarify the structure of a singly linked list.

```cpp
// sll.cpp: singly linked list
// want to: search, insert, delete, maintain sortedness, compute some functio
#include <iostream>
using namespace std;

struct Node {
    Node*  next;
    string payload;
    Node(string pl="", Node* n_ptr=NULL) :
        payload(pl), next(n_ptr) {};
};

void print_list(Node* ptr);

int main() {
    Node* head = new Node("deep");
    head = new Node("the", head);
    head = new Node("in", head);
    head = new Node("rolling", head);
    print_list(head);
    return 0;
}

// print members of the list, starting from 'ptr'
void print_list(Node* ptr) {
    while (ptr != NULL) {
        cout << ptr->payload << " ";
        ptr = ptr->next;
    }
    cout << endl;
}
```

The above program prints `"rolling in the deep"`, and the data structure is illustrated with the following figure:



In general, with a singly linked list we are still wasting $\Omega(n)$ space because of the extra pointer per element. However, this space wasting is not propotional to the size of the items stored in the list as in the vector case. Thus, a linked list typically will require less space overhead than a vector. And, linked list does not suffer from the fragmentation problem.

1.1. Search

Searching for an item in a linked list is inherently a linear-time operation. We just keep going down the list until we find it or encounter `NULL`. (Assuming the list is NULL-terminated. See a problem below!)

```
/**
 * ---------------------------------------------------------------------------
 * search for the first occurrence of name in the list, starting from ptr
 * return NULL if not found, pointer to containing node if found
 * ---------------------------------------------------------------------------
 */
Node* iterative_search(const string& key, Node* ptr) {
    while (ptr != NULL && ptr->payload != key)
        ptr = ptr->next;
    return ptr;
}
```

We can do it recursively too.

```
Node* recursive_search(const string& key, Node* ptr) {
    if (ptr != NULL && ptr->payload != key)
        return recursive_search(key, ptr->next);
    else
        return ptr;
}
```

1.2. Delete

To delete a specific node in a singly linked list which is not the head node, we must have a pointer to the parent node.

```
/**
 * -----------------------------------------------------------------------
 * delete the successor node of the node pointed to by ptr
 * -----------------------------------------------------------------------
 */
void del_successor(Node* ptr) {
    if (ptr == NULL || ptr->next == NULL) return;
    Node* temp = ptr->next;
    ptr->next = temp->next;
    delete temp; // always remember this
}
```

The logic of the code is simple, but if we want to delete the head node we will have to write a slightly different routine; for this reason, many implemenations introduce a dummy sentinel node (http://en.wikipedia.org/wiki/Linked_list#Using_sentinel_nodes) which is never removed. We are not concerned with a clean implementation of the singly linked list data structure in this lecture.

> ***Exercise:*** *write a function that takes a* `string str` *and a* `head` *pointer, finds and deletes the first node with* `str` *payload, and returns a pointer to either NULL or the successor of the removed node.*

The following free all nodes down the list starting from `ptr`

```
/**
 * -----------------------------------------------------------------------
 * free the memory of all nodes starting from ptr down
 * -----------------------------------------------------------------------
 */
void free_list(Node* ptr) {
    Node* temp;
    while (ptr != NULL) {
        temp = ptr;
        ptr = ptr->next;
        delete temp;
    }
}
```

Here is a classic question regarding singly linked lists: given a `head` pointer pointing to the head of a singly linked list (which terminates at NULL), write a function that reverses the list and returns a pointer to the new head node. You should think about it for a little before looking at the solution below.

```cpp
/**
 * -----------------------------------------------------------------------------
 * reverse the list with given head pointer, return pointer to the new head.
 * -----------------------------------------------------------------------------
 */
Node* reverse_sll(Node* head) {
    Node *prev = NULL, *temp;
    while (head != NULL) {
        temp = head->next;
        head->next = prev;
        prev = head;
        head = temp;
    }
    return prev;
}

// and here's to test it
int main() {
    Node* head = NULL;
    ostringstream oss;

    // build a 10-node singly linked list for testing
    for (int i=0; i<10; i++) {
        oss.str(""); // clear buffer
        oss << "Node" << i;
        head = new Node(oss.str(), head);
    }
    print_list(head);
    head = reverse_sll(head);
    print_list(head);
}
```

Exercise: *write a recursive function that reverses a singly-linked list. Your function should take two pointers and return one pointer to* `Node`.

1.3. Insert into a sorted list

A linked list is a pretty good data structure for maintaining a sorted list of items. All we have to do is to go down the list, find the right spot and insert the new node in between. The search takes linear time (in the worse case), but the insertion only takes constant time. This is by contrast to a sorted vector where searching takes logarithmic time but the actual insertion takes linear time in the worst case. The linear time in a vector insert might involve moving large items down the line and thus should be less efficient than the linear search time for a linked list.

```
/**
 * -------------------------------------------------------------------------
 * assume the list is already sorted, insert a new node with the given payloac
 * into the list; return a pointer to the (potentially) new head
 * assume node_ptr points to an allocated node
 * -------------------------------------------------------------------------
 */
Node* insert_into_sorted_list(Node* head, Node* node_ptr) {
    if (head == NULL || node_ptr->payload < head->payload) {
        node_ptr->next = head;
        return node_ptr;
    }

    Node *prev = head, *temp = head->next;
    while (temp != NULL && temp->payload < node_ptr->payload) {
        prev = temp;
        temp = temp->next;
    }

    prev->next     = node_ptr;
    node_ptr->next = temp;
    return head;
}
```

We can test the function in several ways. For example, we could do

```
int main() {
    Node* head = NULL;
    ostringstream oss;

    // build a 10-node singly linked list for testing
    for (int i=9; i>=0; --i) {
        oss.str(""); // clear buffer
        oss << "Node" << i;
        head = new Node(oss.str(), head);
    }

    // print a sorted list
    print_list(head);

    // now delete "Node4"
    Node* temp = search("Node3", head);
    del_successor(temp);
    print_list(head);

    // finally insert "Node4" back
    head = insert_into_sorted_list(head, new Node("Node4"));
    print_list(head);

    return 0;
}
```

Or, we could build a sorted list from scratch.

```
int main() {
    Node* head = NULL, *temp;
    ostringstream oss;
    srand(static_cast<unsigned int>(time(0)));

    // build a 10-node singly linked list for testing
    for (int i=0; i<10; i++) {
        oss.str(""); // clear buffer
        oss << "Node" << (rand() % 10);
        temp = new Node(oss.str());
        head = insert_into_sorted_list(head, temp);
    }
    print_list(head);

    return 0;
}
```

1.4. Compute

There are many other questions we can ask on traversing and computing things with a singly linked list. For example, suppose the `Node` structure holds numbers instead of strings:

```
struct Node {
    Node* next;
    int   payload;
    Node(int pl=0, Node* ptr=NULL) : payload(pl), next(ptr) {};
};
```

are a few examples of questions one might encounter. In what follows we assume that the final node in the list points to `NULL`

1. Given the two heads of two sorted singly linked list, write a function which returns the head pointer to a sorted singly linked list which merges the two given lists.
2. Write a function which takes a head pointer and returns the sum of squares of the payloads.
3. Write a function which takes a head pointer and returns the number of nodes in the list.
4. Swap two sub-blocks of two lists. This is a constant time operation. We cannot do that with a vector/array.
5. Remove consecutive duplicate elements from a singly linked list which is already sorted
6. Remove elements equal to a given value
7. Sort a given list (using insertion sort)

There is one **classic** question which is quite different from what we have seen:

> *In general, in a singly linked list the last node does not have to point to* `NULL`*. It might point back to one of the predecessor nodes. Given a head pointer, write a function which returns* `true` *if the list terminates at* `NULL` *and* `false` *if the list cycles back. Make sure your function uses as little space as possible.*

It is possible to write such a function which takes constant space and linear time.

2. Doubly linked lists and C++ `list` template class

2.1. Doubly linked lists, XOR list

A singly linked list has several disadvantages: (1) we can never traverse backward, (2) we cannot delete a node if we do not have a pointer to the parent node, and (3) we can only insert after a node we have a pointer too. We can fix these disadvantages by having two pointers with a node, one forward and one backward. The price we have to pay is the complexity in code, and the space needed for all the extra pointers.

There is, in fact, one very interesting way to represent a doubly-linked list with only **one** pointer: an XOR linked list (http://www.linuxjournal.com/article/6828). Each node holds the bit-wise XOR of the addresses of the two adjacent nodes. The XOR operation has the nice property that `a^(a^b) = (a^b)^a = b`. Thus, suppose we have three pointers: `prev`, `cur`, and `prev^next` (this is the pointer stored inside the node pointed to by `cur`), then we can compute `next` by doing `prev^(prev^next)`. This way, we can traverse down the list. To traverse backward, we do the reverse.

This idea is rarely used in practice due to code complexity. But, in a small device where memory is precious such as a sensor node, or in some kernel module, I would not be surprised if it is used.

On a related note, here's another classic interview question:

w do you swap two integers without using a third variable?

2.2. C++'s `list` class and `iterator`

The standard template library provides a singly linked list template class called `forward_list` (http://en.cppreference.com/w/cpp/container/forward_list), and a doubly linked list template class called `list` (http://en.cppreference.com/w/cpp/container/list). We navigate a list using an iterator (http://en.wikipedia.org/wiki/Iterator), which is an object that allows programmers to traverse a container (http://en.wikipedia.org/wiki/Container_(data_structure)). A container is an object which is meant to contain other objects. STL's containers include `stack`, `list`, `set`, `map`, `deque`, `forward_list`, and the likes (http://en.wikipedia.org/wiki/Standard_Template_Library#Containers).

An iterator can be thought of as an abstraction for pointers. Iterators allow us to traverse a container and access members of the container. Iterators are directly tight to the internal representation of the container, and thus they are "private data types" of the container. We can access the item that an iterator "points to" by using the dereferencing operator. It is probably best to look at a couple of examples.

```
// iter_list.cpp: navigating a list using iterators
#include <iostream>
#include <list>

using namespace std;

// print all members of the list, assuming << makes sense for T
template <typename T>
void print_list(list<T>& mylist) {
    typename list<T>::iterator it = mylist.begin();
    while (it != mylist.end()) {
        cout << *(it++) << " ";
    }
    cout << endl;
}

/**
 * ------------------------------------------------------------------------
 * main body
 * ------------------------------------------------------------------------
 */
int main() {
    list<int> mylist;

    // build a 10-node linked list for testing
    for (int i=0; i<10; i++) {
        mylist.push_back(i);
    }
    print_list(mylist);

    return 0;
}
```

The keyword `typename` has to be put before defining the variable `it` because we have to tell the compiler (http://womble.decadent.org.uk/c++/template-faq.html#type-syntax-error) that `list<T>::iterator` is a type that's dependent on the template parameter.

There are two methods `erase()` and `remove()` for deleting elements of a `list`. `erase` can be used to remove an element pointed to by an iterator. After erasing, the iterator becomes invalid (think: the pointer points to a memory location which was freed). However, the `erase` function returns an iterator to the next element down the list; hence, if you want to know where we are after erasing, just do `it = mylist.erase(it)`. The `remove` method removes from the list all elements equal to a given value.

We don't have to physically reverse the list. We can use a reverse iterator (http://www.cplusplus.com/reference/std/iterator/reverse_iterator/) to iterate from the end:

```cpp
template <typename T>
void print_list(list<T>& mylist) {
    // typename list<T>::iterator it = mylist.begin();
    typename list<T>::reverse_iterator it = mylist.rbegin();
    while (it != mylist.rend()) {
        cout << *(it++) << " ";
    }
    cout << endl;
}
```

A reversed iterator is an iterator where the meaning of the operators `+, -, ++, --` and the likes are reversed. We should start from `rbegin()` which is the iterator to the end of the list. We cannot start from `end()` since `mylist.end()` has the semantic of a `NULL` pointer.

2.3. Implementing a `UBList` class

Implementing a list class is not difficult. The technical hurdle is mostly syntactical because we will have to define the iterator types ourselves. Our textbook has a section on implementing doubly linked list. And, the TAs will explain that implementation in some details.

3. Implementing stacks and queues with lists

d lists are excellent choices for the underlying data structure for stacks and and queues. We have stacks before, which is a container where elements are inserted and accessed in a last-in-first-out (LIFO) order. A queue is a container where elements are accessed and inserted in a first-in-first-out (FIFO) order. C++ provides a double-ended queue (`deque`) where elements can be accessed and inserted from both ends.

In any case, it should be obvious why linked lists are a good choice for implementing stacks and queues.

4. Skip lists

Going beyond stacks and queues, let us consider other operations that we may want to perform with a sorted (doubly) linked list:

```
Operation                               Time Complexity
-----------------------------------------------------
Search                                  O(N)
Insertion                               O(N)
Removal                                 O(N)
Insertion and removal from the ends     O(1)
Insertion and removal a given node      O(1)
```

The run times for insertion and removal are pretty horrible, which is dominated by the search time. In a sorted vector we can do search in $O(\log n)$ time. Can we design a linked list with better search time? In 1990, Professor William Pugh of Maryland proposed a very cute data structure called skip list (ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf) that can achieve the desired $O(\log n)$ search time. If we can search in $O(\log n)$ time, then we can do insert and delete in the same asymptotic time because after searching we have a pointer to the node (assumming we don't have to update too many pointers after insertion or deletion of a given node).

4.1. Search only

Let's first assume we do not do any insertion or deletion. The idea of a skip list is very simple. We add ional pointers to the nodes in the list so that we can jump quicker into the middle of the list. Say we n elements in the list. At level `0`, each element has a `next` pointer as usual. At level `1`, element ›ered `2k` has a pointer to element numbered `2(k+1)`. At level `2`, element numbered `4k` has a pointer to element numbered `4(k+1)`, and so forth. This way, we have about $O(\log n)$ levels as the following picture illustrates. (Picture taken from this link (http://msdn.microsoft.com/en-us/library/ms379573(v=vs.80).aspx).)

One more level

- To search for an element `key`, we find two consecutive elements at the top level where `key` is in between and then move down one level and repeat. The search obviously takes $O(\log n)$-time, much as binary search.
- How much is the storage overhead we have to pay? At the `i`th level, there are $\frac{n}{2^i}$ extra pointers. Thus, overall we are wasting at most a big-O of
  $\sum_{i=0}^{\log n} \frac{n}{2^i} \leq n\left(\sum_{i=0}^{\infty} 1/2^i\right) = 2n = O(n)$

  of space overhead. The space overhead is still `O(n)` in terms of pointers (as opposed to `O(n)` in terms of the item sizes). A skip list wastes about twice as many pointers as a normal linked list, but it allows for searching for a given element in time $O(\log n)$. This is a very nice feature to have at a small price.

4.2. Insert and delete

Inserting into a skip list will destroy the balancing structure of the list as described in the above idealized situation. If we insert or delete too many elements, we will probably have to re-organize the list so that it becomes “balanced”, and balancing a skip list takes `O(n)` time. There are two choices:

1. we only rebalance the list once in a while and amortize that cost to the operations in between
2. build into each operation (insert/delete) some structure so that the list still has the nice $O(\log n)$ search time without re-balancing.

The recomended option (http://drum.lib.umd.edu/bitstream/1903/544/2/CS-TR-2286.1.pdf) is option 2. The idea is also extremely simple: when we insert a new element into the list, we first find (hopefully in logarithmic time) the correct place at level 0 to insert the new element. Then, we flip a coin and if it comes up head the element will “float up” to the next level. If it does float up, we flip another coin, and so forth, until some `max_level` is reached. On average, about half of the elements will float up one level, a quater two levels, one eighth three levels, and so on. Thus, similar to our coin-flipping in the quick sort algorithm, the skip list will work very well on average (http://msdn.microsoft.com/en-us/library/ms379573).

> ***Programming project:*** *implement a skip list of integers with the randomized insertion and deletion strategy discussed above. Insert randomly 50K integers in to the list. Then, perform 50K random searches. Compare the total run time of insertion and search with our UBVector class.*

◎ C++ iterators, Linked lists, Skip