A doubly linked list

Fatima Mohammad Ali

Fall 23/09/2023

Introduction

The focus of this assignment was to explore *doubly linked lists* by implementing several useful methods and compare it to singly linked lists. The comparison was done by analysing how both structures behave when *unlinking* and *inserting* certain elements in growing sizes of lists.

Doubly linked list

A doubly linked list is a list of *cell* objects. Each cell holds an integer value, a *previous pointer* (`Cell prev`), and a *forward pointer* (`Cell next`). The list has access to the *first* cell (`Cell first`) to navigate through the sequence by updating each cell's references.

Several methods were implemented to expand the doubly linked structure. To add an item at the beginning of a list, an `add(int item)` function was used. A new cell with the given item value was initialized as a start. If the list was not empty, the `next` pointer of the new cell would point to whatever `first` was pointing at, and its `prev` pointer would be set to null (since it will be the first cell in the list). The `first` cell would then be set to point to the new cell. Thus, a new cell was added to the beginning of the sequence.

The two methods `find(int item)` and `length()` were also implemented. Both methods solely utilized the `next` pointer of a cell to traverse through the list and carry out their respective function by checking and returning relevant conditions and values. Their implementation was similar to that of a singly linked list.

Additionally, a `remove(int item)` method that removes an item from the list was coded. Unlike `remove(int item)` in singly linked structures where a "previous cell" was initialized and used, a doubly linked cell already has a "`prev`" pointer that points to the previous cell. After traversing a doubly linked list and finding the cell with unwanted value, it was *unlinked* from the list in a procedure that is further explained in the section below.

Unlink method

If there is a need to remove a *specific cell* from a list, it could be *unlinked* by updating its two references, `prev` and `next`, to neatly cut it off from the rest of the sequence.

Unlink in a doubly linked list

Contrary to `remove(int item)`, the following `unlink(Cell del)` method do not need to iterate through the list to find a specific cell since the given cell is already "holding" its "location" by its references. The unlinking procedure was implemented as follows (the `remove` method also used this procedure to remove a cell with unwanted value after finding it):

```
public void unlink (Cell del){
    //if del isn't the last cell
    if(del.next != null)
        del.next.prev = del.prev;

    //if del isn't the first cell
    if(del.prev != null)
        del.prev.next = del.next;
    else  //if del is the first cell
        first = del.next;

    //nullify the deleted cell's pointer
    del.prev = null;
    del.next = null;
}
```

When given a cell (`del`), the method would check three cases. If `del` was not the last cell, the next cell's `prev` would point to whatever `del`'s `prev` pointer was pointing at. If `del` was not the first cell, the previous cell's `next` pointer would point to whatever `del`'s `next` pointer was pointing at. If `del` was the first cell, it would be skipped by `first` directly pointing to the `next` pointer of `del`. Thus, the unwanted cell was unlinked from the list.

Unlink in a singly linked list

Since there is no previous pointer in singly linked cells, a `Cell prev = null` was used to mimic a `prev` pointer. The `prev` cell was constantly incremented one step behind the *current* cell when traversing through the list. When the unwanted cell was found, the `prev` cell would point to the next cell (`prev.next = cur.next;`) to skip over the current unwanted cell (similarly to how it is *removed* from a singly linked list in the previous assignment).

Insert method

The main approach of inserting a cell at the beginning of a list is to link it to the first cell, similarly to how it was added in `add(int item)`. The big difference here is that a cell is given as parameter.

Insert in a doubly linked list

Inserting a cell at the beginning of a doubly linked list was done by linking the given cell to the `first` cell. Two cases were considered, if the list was empty or non-empty. The `insertFirst(Cell item)` was implemented as follows:

```
public void insertFirst(Cell item){
    //if the list is non-empty
    if(first != null){
    //new cell's next pointer will point at whatever first is pointing to
        item.next = first;
    //prev pointer of first will point to the new cell
        first.prev = item;
    //prev pointer of the new cell is null
        item.prev = null;
    }
    //first only points to the new cell
    first = item;
}
```

Insert in a singly linked list

Similarly to its `add(int item)` method (that was implemented in the previous assignment), inserting a cell to the beginning of a singly linked list was done by checking two cases. If the list was not empty, the `next` pointer of the new cell would point to whatever `first` was pointing at and `first` would be set only pointing at the new cell. If the list was empty, the new cell would directly be set as `first` and hence inserted at the beginning.

Benchmark implementation

The benchmark in this task checked the required time to unlink and insert 1000 randomly selected cells in singly respective doubly linked lists. After allocating a list of n cells, their references were kept in an n sized *cell array*. To access cells at 1000 random indices of the cell array, an array of k keys (with numbers between 0 and *n-1*) was used. The following code snippet shows a part of the benchmark of a singly linked list (listSLL) of size n.

```
for (int i = 0; i < tries; i++) {   //tries = 100
    double t0 = System.nanoTime();
    for(int j = 0; j < k; j++){ //k random keys, k = 1000
        listSLL.unlink(cellarray[keys[j]]);
        listSLL.insertFirst(cellarray[keys[j]]);
    }
    double t1 = System.nanoTime();
    :
}
```

Results

Table 1 shows the benchmark results of growing singly respective doubly linked lists.

n	singly LL	Doubly LL
200	150	7.6
400	280	6.2
800	500	4.4
1600	900	8.6
3200	1900	5.7
6400	2500	5.7
12800	3200	6.6
25600	4000	7.3

Table 1: Timestamps are shown in microseconds where 1000 randomly found cells were unlinked and inserted. Each timestamp is the minimum value of 100 tries.

Discussion

As expected, the doubly linked list is significantly faster than its counterpart thanks to the use of its two pointers. The singly linked structure has to constantly iterate through the list to find a specific cell and unlink it. At worst case, it would have the big-O notation $O(n)$ where n is the number of iterated cells/size of list. A doubly linked structure however, would unlink a cell in constant time ($O(1)$) since every cell has its "position" in the list already given through its references. This could be seen in table 1 where unlinking and inserting elements is constant (approx. 6 microseconds) for doubly lists, while it is growing for single structures.