Performance Study Report on 3 different Data Structures

Terry Cheong (@daydreamer)
張顥霆 電機二 B03901175

November 26, 2015

1 Introduction

In this report we will discuss about the design, implementation and performance of 3 data structures: dynamic array, circular doubly linked list, binary search tree, together with some basic analysis of time complexity of each operation.

2 Design and Implementation

2.1 Dynamic array

Dynamic array is a randomized access, variable size data structure that allows inserting and deleting elements. It has a initial capacity, which will grow with insertion of new elements.

The data structure is implemented by a structure having a array and two integers that records the capacity and size of the current array.

2.1.1 Operations

Getting number of elements

As the size variable is maintained during the insertion and deletion process, we can get the size of array in $O(1)$ time.

Insertion at the end

Inserting an element at the end of a array is pretty easy: adding it to the back and add the size by 1. In this case, this operation only cost a constant time, a.k.a. $O(1)$ time. In case that the size is larger or equal to the capacity of the array, we reallocate the memory of the array with a capacity that is double of the previous one and copy the elements. Thus, this operation will cost $O(n)$ time in this case, where n is the size of the array.

In average, while inserting n elements into a array of size n, the amortized cost of the insertion process is about $O(n/n) = O(1)$ as the reallocate process only done once.

Deletion

Deletion an arbitrary instance in an array of size n is implemented as the following:

1. delete the specified instance
2. shift the subarray that started from the element after the specified instance to the end of the array to left by 1.
3. Decrease the size variable by 1.

Thus, the time cost of the operation is $\Theta(n)$ as the shifting step need to access each element in the subarray once. There is a special case of deletion: deleting the last element,a.k.a. pop back, costs $O(1)$ time.

Sorting

Sorting an array of size n is implemented by calling the sort function in C++ STL(Standard Template Library). This step has a cost of $\Theta(n \log n)$

Iteration

Iterator class is introduced to travel through the whole array. The class is a wrapper of a pointer to element. The class has two function: $++$ and $--$, which is accordingly step forward and step backward. Each call move the iterator by one step.

As moving in an array only need to do arithmetic operations to the pointer, moving one step for the iterator cost $O(1)$ time.

2.2 Circular doubly linked list

Linked list is a sequence of nodes that are connected by having a link from a node to the next. Circular doubly linked list is a extension of a linked list where each node is having two link to its predecessor and successor respectively. Also the last element of the list will point to the first element in the list. In here we added a dummy node that will help in implementing some operations.

The data structure is implemented by defining a node containing data and two pointers, pointing to the next node and the previous one. Using the node structure, the list structure has a pointer that points the dummy node. The start of the list is the next node of the dummy node. Noting that the list is initialized that the previous node and the next node of the dummy node is the dummy node itself.

2.2.1 Operations

Getting number of elements

In order to get the number of elements, we have to go through the whole list once. Therefore this operation will have a time cost of $O(n)$, where n is the size of the list.

Insertion at the end

The end of the list can be accessed by calling the previous node of the dummy node. Therefore, the insertion process is implemented by adding a new node as the successor the original end node and point to the dummy node. This process should reset the pointers of the dummy node and the original end node such that the pointers points to the correct node.

This process only need to do constant times of pointer operations. Thus, this operation have a time cost of $O(1)$.

Deletion

Random deletion in circular doubly linked list is pretty easy: remove the specified node and then reset the pointers of the predecessor and successor of the node such that the latter node become the next node of the former node. This will have a time of $O(1)$ as only constant time of pointer operation is needed.

Sorting

Merge sort is implemented to sort the linked list structure for a much better performance. There are two helper functions implemented: FindMid, Merge. The FindMid function return the midpoint of the list, by having two iterators that move one step forward and two step forward accordingly. When the faster one reach the end, terminate the function and return the slower iterator. Then we can split the list into two parts by the node that the iterator points to.

After recursively sorting the two parts of the list, we use the merge function to merge them together. This is implemented in a tricky way in the code. But the idea is simple:

First of all we have two pointers that points to the beginning of the two parts. For each step we compare the data that the pointers point to, and append the node that have the smaller data to the sorted list and move the corresponding pointer one step forward. When one of pointers reach its end, assume that the data it points to is infinity large. Repeat the steps and we will have a sorted list.

This operation have a time complexity of $\Theta(n \log n)$ by solving the recurrence relation $T(n) = 2T(\frac{n}{2}) + \Theta(n)$.

Iteration

Similarly, iterator class is implemented in the linked list structure. As moving the iterator by one step only needs to access the corresponding pointer, this operation will cost $O(1)$.

2.3 Binary Search Tree

Binary search tree is a binary tree structure that has the property that the data in the left child is less than the data in the current node and the one in the right child is greater than the data in the current node. Therefore we can get the sorted order of elements easily thanks to this property.
In my implementation, each node contains a data and 3 pointers that points to its parent, left child and right child. The tree itself contains a size variable and 2 pointers, one points to the root and one points to the tail node. Here tail node is a dummy node that have a value infinity large for convenience of iteration.
The tree is initialized by having a dummy node in the root and have a size 0.

2.3.1 Operations

Getting number of elements
As we do maintain a size variable, getting the number of elements will cost a $O(1)$ time.

Insertion
Different from the two structure introduced, insertion in binary search tree don't have a position as the position itself is determined in order to maintain the property that is introduced before.

The insertion procedure starts with examining the root node. Compare the data that need to be inserted with the data of the current node, go examine the right subtree if the current node smaller, otherwise the left subtree. Repeat the process until we reach a leaf node or the dummy node. If it is the dummy node, replace the dummy node with a new node and let its right child be the dummy. For the other case, we add the new node as the child of the leaf node such that the property is satisfied.

Finally we need to add up the size variable by 1. This will have a average time cost of $O(\log n)$, as on average a binary tree of size n with have a height of $O(\log n)$.

But we should know that when the tree is inserted with a sorted sequence, the tree will have a height of size n and thus the operation in this case have a time cost $O(n)$.

Iteration
Similar iterator class as the previous structures is introduced but something is different in the implementation of the iterator class.

First, we start the iteration from the smallest element in the tree, which can be accessed by keep going to the left subtree and ends when we reach the dummy node.

The process of finding the successor of a node is as the following

1. If the node has a right child, return the smallest node in its right subtree.
2. If not, keep going to it's parent until the node is a left child of its parent.

Similarly we can find the predecessor of a node.

The operation of moving the iterator by one step cost $O(\log n)$ time as the average height of the tree is $O(\log n)$.

Deletion
Deleting a specified node in a binary search tree need to consider 3 cases:

1. If it is a leaf node, simply remove it.
2. It it has only one child, remove it and replace it with its child.
3. If it has two children, replace the data in the node with the data in its successor node R, and recursively delete the node R.

And of course, decrease the size by 1.

This operation have a time cost of $O(\log n)$ as sometimes it have to find the successor, and this will traverse the node with a number which is around the height of the tree.

Sorting
Sorting is not needed as the iteration will iterate in a sorted order.

2.4 Comparisons

Below is the table that compares the time complexity of the operations in the 3 structures.

	Array	DList	BST
Insertion	$O(1)$	$O(1)$	$O(\log n)$
Deletion	$O(n)$	$O(1)$	$O(\log n)$
Sorting	$O(n \log n)$	$O(n \log n)$	N/A

While iterating in the BST is a much harder job, the other two structures have a much easier time iterating through the array. But only the array structure can be random accessed and therefore random deletion in the program is much faster than the other two structures as the random indicator can be generated easily.

3 Performance Tests

Performance test is done on 3 different operations: random adding, random delete, sorting. The test environment is:

- CPU: Intel(R) Core(TM) i5-4200U CPU 1.60GHz
  RAM:3956952 kB
- Compile arguments: -O3 -Wall -DTA_ KB_ SETTING -$(PKGFLAGS)
- OS: 4.2.5-1-ARCH x86_ 64 GNU/Linux

The testing scheme is to do the operation and record its data. Repeat 5 times and take their average as the final value.

Also, due to short of time, special case studies are not done in this report.

3.1 Random insertion

The table shown below gives the comparison of the time needed by operation with different input size. We can see that the BST is performing the worst in inserting, just as we analyzed, while Dlist is slightly faster than array.

N	Array	Dlist	BST
10^4	0.01	0.005	0.009
3×10^4	0.01	0.008	0.014
10^5	0.048	0.012	0.088
3×10^5	0.186	0.054	0.358
10^6	0.448	0.192	1.52
3×10^6	1.622	0.58	5.606
10^7	6.118	1.94	22.694

Table 1: Performance comparison for random insertion in second

3.2 Random deletion

The table below show that time in seconds that each structure takes to delete N nodes in random. We cannot do a large size as it takes over 5 minutes for $N = 10^5$. We can see that the random access of array accelerate the deletion in array in larger size. This is resulted from the way random iterator is generated. The BST is also the slowest in the 3 structures.

N	Array	Dlist	BST
10^4	0.266	0.18	0.668
3×10^4	2.41	2.454	8.622
10^5	33.328	44.35	228.74

Table 2: Performance comparison for random deletion in second

3.3 Memory Usage

The table below shows that the usage of memory of each data structure in different size N.
We can see that the array size is always larger, as we allocate a new block of memory which is the size of previous one, which means we will have many unused space when N is not close to power of 2.

N	Array	Dlist	BST
10^5	6.5856	3.840	5.3944
3×10^5	38.094	19.072	23.616
10^6	80.068	72.402	87.644
3×10^6	332.08	225.02	270.64
10^7	1340	759.04	911.5

Table 3: Performance comparison for memory usage in MB