B03901034 吳泓霖

DSnP HW5 Report

I. Introduction

1. Array:

Arrays store data continuously in memory. Traditional arrays have fixed size, our goal is to create a dynamic array, which can handle any amount of data in runtime. Because array stores data continuously, it has the following characteristics: we can access every element in constant time, so data in arrays are random accessible; it requires a large block of continuous memory, so if the data number exceeds the capacity, it has to reallocate a new space, copy the old data, then delete the previous memory; remove/insert data in positions apart from the end requires shifting data, which costs a lot of running time.

2. DList:

Lists do not store data continuously in memory. Instead, it uses pointers to link data together as a list. In our case, we use “doubly linked list”, which means in every node, we store the pointer pointing to the next and previous node. We won’t know the location of a certain node in the list, unless we travel from the head to our target. Accessing certain data requires much more time than array does, so it is not random accessible.

3. BST:

Binary search trees also don’t store data continuously. The difference from lists is that BSTs don’t simply link data together as a list, but as a special tree with rules. The cleverly organized tree results in many benefits. Inserting, deleting, and accessing data all requires running time O(logn). Although each is not the best, BSTs give a very good running time for all functions, which may be useful for situations that we need all functions not to run too slow. Also, the data are sorted at the time it was pushed into the tree.

II. Implementation

1. Array

Members:

In the class Array, we have private member _data, _size, and _capacity. _data points to the array we store data, _size records how many data are stored in the array, _capacity records how large the array is. _capacity changes from 0 to 1, 2, 4, 8……, which means that we will double the capacity every time the data exceeds the current capacity.

Iterator:

Implementing iterator is easy. We can directly ++, --, +, - to _node.

Functions:

I use a private function *expand()* to handle the capacity expansion, it will first create a new array with the size of the new capacity, then copy all data one by one in the old array to the new one, finally delete the old array to release memory.

begin() returns iterator(_data), it's the first element. *end()* returns iterator(_data+_size), it's the next position of the last element.

For *push_back()*, we check the capacity and expand if needed, assign the data to index number [_size], then increase _size.

There are several functions which will delete objects: *pop_front()*, *pop_back()*, *erase(iterator)*, *erase(data)*. Actually I only implemented the function *erase(iterator)*, others are all using this function to delete data. For *pop_front()*, I call *erase(begin())*; for *pop_back()*, I call *erase(--end())*; for *erase(data)*, I first scan through the array to find the position of data, then call *erase(position)*. Everything in this paragraph is also true in DList and BST, so I don't have to explain again.

For *erase(iterator)*, I shift all data after the iterator position one index to the front, then decrease _size. For *clear()*, it's unnecessary to remove the data from the array, we just have to make it inaccessible to users, so simply just set _size to 0 is much faster than clearing the data one by one.

2. DList

Members:

In the class DList, we have private member _head. At the beginning, I decided that _first should point to the first element of the list. But this created some inconvenience for my implementation, because:

a. I have to maintain _head in several functions which is a bit annoying, and most importantly

b. *sort()* is a constant function, which means that _head cannot be changed, and my sorting method is much easier to change pointers instead of the data itself . So at last I settled to let _first always points to the dummy node. By doing so, _head will never have to be changed once DList is initialized. The dummy node is used as the last element in list, but in my case, it has a much more important usage. It can be seen like this: the whole list is an unknown path stored in heap memory, _head (which points to dummy) is the only key to open the door of this path, and the key will not change over time.

Iterator:

For (++/ --), we assign _node to _node->(_next/_prev).

Functions:

I have 3 private helper functions: *insert(data, pos)*, *insert(node, pos)*, and *pop(pos)*. Note that pos and node are iterators. *insert(data, pos)* will create a new DListnode with data, and place it in front of pos. *insert(node, pos)* will place node in front of pos. *pop(pos)* will detach pos from the list.

For *begin()*, I return iterator(_head->_next). For *end()*, I return iterator(_head).

For *push_back()*, I call insert(data, end()). This will place a new DListnode with data included in front of dummy. For *erase(pos)*, I'll first call *pop(pos)*, then delete pos. For *clear()*, I scan through the whole list and delete them one by one, this is faster than calling *erase()* to all elements because *erase()* will also handle the linking pointers, which in this case is unnecessary.

I've implemented two different sorting algorithm. Both can be called in *sort()*, but only choose one to use. For *insertion_sort()*, starting from the second node and all the way to the last, scan through all previous nodes of the current node and find the place to insert it. I use my helper functions

pop(), insert() to accomplish pulling the node out and insert it to the correct position. Insertion sort has running time O(n^2) in theory, but has a very good running time if the list is nearly or already sorted.

For *quick_sort(start, end)*, things become a little tricky. I call the list between start and end the “sublist”. The procedure can be split to following steps:

a. Find the pivot. To avoid the worst case of quicksort, I choose the median of start, end, and “the center node in the sublist” as the pivot.
b. Split the sublist into two parts: left side with elements smaller than the pivot, right side with elements larger than or equal to the pivot. To accomplish this, we scan from start to end, end to start at the same time, and swap the nodes that are not positioned correctly.
c. Insert the pivot back to the sublist in the position where the scanning in part b. terminates. Then recursively quicksort the left side of the pivot and the right side.
d. Special cases: start and end are neighbors, then just compare and sort it; start and end are the same, then do nothing.

Quicksort has average running time O(nlogn) in theory, but has a worst case which is O(n^2). I avoided this case in step a.

3. BST

Node:

The class BSTreeNode has member _data, _parent, _left, and _right. _data is the data, parent points to the parent of the node, _left/_right points to the left/right child. The constructor of BSTreeNode accepts 4 parameters, the data, parent, left/right child. The parent, left/right child is set to 0 (null pointer) by default.

Member:

The class BSTree has member _root, _dummy, and _size. _root points to the root node of the tree, _dummy points to the dummy node. The dummy node is always maintained to the rightmost node of the tree, which means the right child of the largest node. It is for convenience to return the *end()* iterator. _size will record the size of the tree.

Iterator:

The iterator is important because it handles the traversal of the tree. For ++, first check if it has a right node. If it has, the result is the leftmost node of its right node. If it has no right node, then the result is the smallest left child in its ancestors. For --, it is nearly the same as ++, but swap left and right in the procedure.

Functions:

I defined some macros for helping. *TOMAX(A)* and *TOMIN(A)* will go to the rightmost/leftmost node of node A. *REPLACE(A, B)* (B has to be a child of A) will replace node A with B, it will determine whether A is a left child or right child, and handle all linking pointers.

For *begin()*, return iterator *TOMIN(_root)*. For end(), return iterator(_dummy), since the dummy is always maintained as the next node of the largest node.

For *insert(data)*, starting from the root, it will go left/right if the data is smaller/(larger or equal) than the current node. It will continue to go through the tree until it meets a NULL pointer, then it will place itself in that position.

For *erase(pos)*, it contains 3 situations:

a. No child: Call *REPLACE(pos, NULL)*, then delete pos.
b. One child: Call REPLACE(pos, left/right child of pos), then delete NULL.
c. Two children: Copy the data of the next node of pos to pos, then call *erase(next node of pos)*. The next node of pos will always be one of the situation of a./b., so it will work.

Note that we have to maintain _dummy. In case a, pos has _dummy as its right child, call *REPLACE(pos, _dummy)* and delete pos. In case b, pos has left child and _dummy is its right child. _dummy has to connect to the next node of pos, then we can delete pos.

For clear(), we erase every node from *begin()* to *end()*.

III. Performance

1. Design

I create a dofile with the following:

adta -r 100000; usage; adtp; usage; adts; usage; adts; usage;

adtd -f 100000; usage; adta -r 100000; usage; adtd -b 100000; usage;

adta -r 100000; usage; adtd -a; usage; q -f;

The **1st** usage will show how it handles **insert**. The **2nd** usage will show how the **traversal** perform. The **3rd** usage will show how it handles **sort**. The **4th** usage will show how it handles **already sorted** data. The **5th** usage will show how it handles **pop front**. The **7th** usage will show how it handles **pop back**. The **9th** usage will show how it handles **clear**.

2. Expectation

Array:

All procedures should perform really great, except the forth procedure, it'll take a long time to pop all the data from the front, because every pop front requires shifting the whole array. It will require the most memory because it has to pre-allocate a large block of data for storing data continuously.

DList:

All procedures should perform really great, except the second and third procedure, which depends on the algorithm.

BST:

All procedures should perform fairly well, but not the best.

3. Results (The number in front is the number of usage)

Ref-Array:

1
```
Period time used : 0.02 seconds
Total time used  : 0.02 seconds
Total memory used: 6.094 M Bytes
```

2
```
Period time used : 0.16 seconds
Total time used  : 0.18 seconds
Total memory used: 6.094 M Bytes
```

3
```
Period time used : 0.04 seconds
Total time used  : 0.22 seconds
Total memory used: 6.098 M Bytes
```

4
```
Period time used : 0 seconds
Total time used  : 0.22 seconds
Total memory used: 6.098 M Bytes
```

5
```
Period time used : 72.26 seconds
Total time used  : 72.48 seconds
Total memory used: 6.098 M Bytes
```

7
```
Period time used : 0 seconds
Total time used  : 72.49 seconds
Total memory used: 6.098 M Bytes
```

9
```
Period time used : 0 seconds
Total time used  : 72.49 seconds
Total memory used: 6.098 M Bytes
```

My-Array:

1
```
Period time used : 0.03 seconds
Total time used  : 0.03 seconds
Total memory used: 6.105 M Bytes
```

2
```
Period time used : 0.15 seconds
Total time used  : 0.18 seconds
Total memory used: 6.109 M Bytes
```

3
```
Period time used : 0.04 seconds
Total time used  : 0.22 seconds
Total memory used: 6.113 M Bytes
```

4
```
Period time used : 0 seconds
Total time used  : 0.22 seconds
Total memory used: 6.113 M Bytes
```

5
```
Period time used : 70.31 seconds
Total time used  : 70.53 seconds
Total memory used: 6.113 M Bytes
```

7
```
Period time used : 0 seconds
Total time used  : 70.54 seconds
Total memory used: 6.113 M Bytes
```

9
```
Period time used : 0 seconds
Total time used  : 70.55 seconds
Total memory used: 6.113 M Bytes
```

Ref-DList:

1
```
Period time used : 0.01 seconds
Total time used  : 0.01 seconds
Total memory used: 4.723 M Bytes
```

2
```
Period time used : 0.17 seconds
Total time used  : 0.18 seconds
Total memory used: 4.723 M Bytes
```

3
```
Period time used : 143.4 seconds
Total time used  : 143.6 seconds
Total memory used: 4.723 M Bytes
```

4
```
Period time used : 35.81 seconds
Total time used  : 179.4 seconds
Total memory used: 4.723 M Bytes
```

5
```
Period time used : 0.01 seconds
Total time used  : 179.4 seconds
Total memory used: 4.723 M Bytes
```

7
```
Period time used : 0.01 seconds
Total time used  : 179.4 seconds
Total memory used: 5.5 M Bytes
```

9
```
Period time used : 0.01 seconds
Total time used  : 179.5 seconds
Total memory used: 5.5 M Bytes
```

My-DList (Insertion Sort):

1
```
Period time used : 0.01 seconds
Total time used  : 0.01 seconds
Total memory used: 4.719 M Bytes
```

2
```
Period time used : 0.16 seconds
Total time used  : 0.17 seconds
Total memory used: 4.719 M Bytes
```

3
```
Period time used : 54.56 seconds
Total time used  : 54.73 seconds
Total memory used: 4.719 M Bytes
```

4
```
Period time used : 0.01 seconds
Total time used  : 54.75 seconds
Total memory used: 4.719 M Bytes
```

5
```
Period time used : 0.01 seconds
Total time used  : 54.75 seconds
Total memory used: 4.719 M Bytes
```

7
```
Period time used : 0 seconds
Total time used  : 54.77 seconds
Total memory used: 5.637 M Bytes
```

9
```
Period time used : 0.01 seconds
Total time used  : 54.79 seconds
Total memory used: 5.637 M Bytes
```

My-DList (Quick Sort):

1
```
Period time used : 0.01 seconds
Total time used  : 0.01 seconds
Total memory used: 4.719 M Bytes
```

2
```
Period time used : 0.17 seconds
Total time used  : 0.18 seconds
Total memory used: 4.719 M Bytes
```

3
```
Period time used : 0.05 seconds
Total time used  : 0.23 seconds
Total memory used: 4.719 M Bytes
```

4
```
Period time used : 0.01 seconds
Total time used  : 0.24 seconds
Total memory used: 4.719 M Bytes
```

5
```
Period time used : 0.01 seconds
Total time used  : 0.25 seconds
Total memory used: 4.719 M Bytes
```

7
```
Period time used : 0.01 seconds
Total time used  : 0.27 seconds
Total memory used: 6.539 M Bytes
```

```
Period time used : 0.01 seconds
Total time used  : 0.3 seconds
Total memory used: 6.848 M Bytes
```

Ref-BST (3, 4 is ignored):

```
Period time used : 0.05 seconds
Total time used  : 0.05 seconds
Total memory used: 4.719 M Bytes
```

```
Period time used : 0.29 seconds
Total time used  : 0.34 seconds
Total memory used: 4.723 M Bytes
```

```
Period time used : 0.02 seconds
Total time used  : 0.36 seconds
Total memory used: 4.723 M Bytes
```

```
Period time used : 0.03 seconds
Total time used  : 0.43 seconds
Total memory used: 6.234 M Bytes
```

```
Period time used : 0.01 seconds
Total time used  : 0.49 seconds
Total memory used: 6.996 M Bytes
```

My-BST (3, 4 ignored)

```
Period time used : 0.05 seconds
Total time used  : 0.05 seconds
Total memory used: 4.715 M Bytes
```

```
Period time used : 0.16 seconds
Total time used  : 0.21 seconds
Total memory used: 4.715 M Bytes
```

```
Period time used : 0.02 seconds
Total time used  : 0.23 seconds
Total memory used: 4.715 M Bytes
```

```
Period time used : 0.03 seconds
Total time used  : 0.3 seconds
Total memory used: 5.484 M Bytes
```

```
Period time used : 0.02 seconds
Total time used  : 0.36 seconds
Total memory used: 5.746 M Bytes
```

Discussion:

Array: As expected, all procedures are really fast, except pop front. Insert is slightly slower because it has to expand capacity at runtime.

DList: As expected, all procedures are really fast. The reference program might use bubble sort, so the sorting time is O(n^2), even if it is already sorted. Insertion sort has better running time, and sorting sorted data only requires O(n). The improved quicksort has very good performance in any conditions, all is O(nlogn).

BST: As expected, all procedures are fast, but not the best. It provides a really stable performance, every step requires O(logn).

So the winner in this contest should be my “Quick-sort implemented DList”. It has nearly every function running as the best. However, implementing quicksort on DList isn’t an easy task, so if we don’t need deleting/inserting data to the front very often, it’s better to use array, because it’s easy to implement, and also have really great performance.