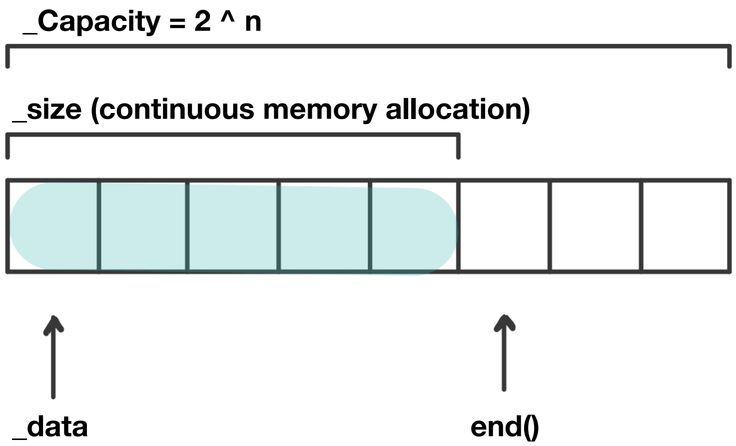
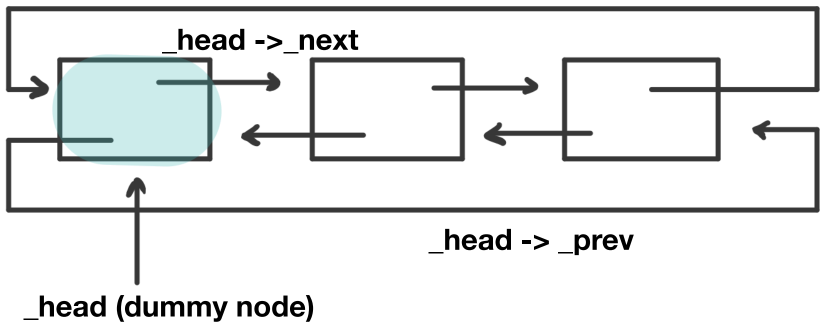
Data Structures and Programming HW5. Report

B06901601 王廷峻

I. Array

Dynamic Array		
Illustration of array		
	Public member function	
void push_back (const T&)	**Case1 : _size < _capacity** Add the entry to the last position of the array. **Case2 : _size = _capacity** Call helper function : expand() and case2 reduces to case 1.	O (1)
bool erase (iterator)	Replace the unwanted element with the last entry of an array. ‣ Pros : It makes deletion as constant time algorithm, instead of O(N) when requiring to maintain the order of an array. ‣ Cons : Doesn't preserve the order of a sorted array.	O (1)
iterator find (const T&)	Linear search from the first element to the end Return the iterator pointing to the wanted element or return end() if not found. ‣ Pros : Easily to implement. ‣ Cons : Doesn't take advantage of sorted array (binary search O(logN)).	O (N)
void sort () const	**Case1 : _isSorted** Do nothing **Case2 : NOT _isSorted** STL sorting	O (NlogN)
	Auxiliary member function	
void expand ()	Create an array with double capacity and copy each element of the previous array to the newly created array.	O (N)

II. Doubly-linked List

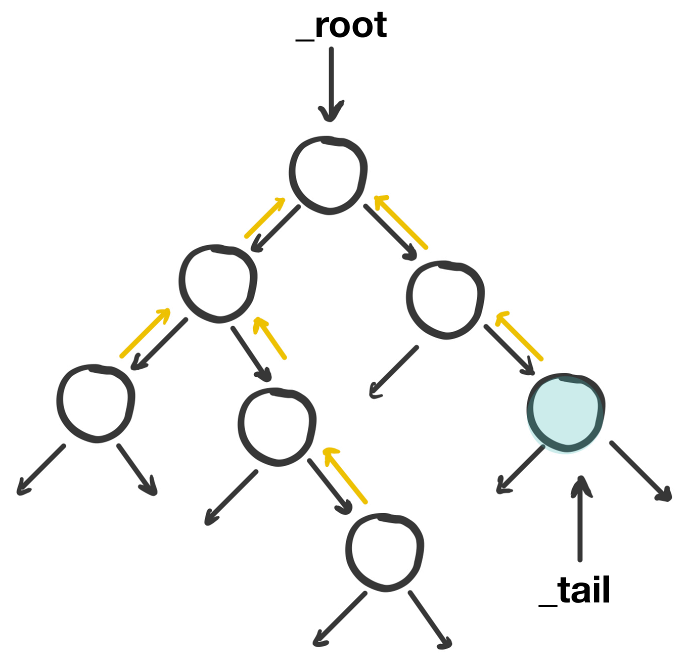
Doubly-linked list		
Illustration of doubly-linked list		
	Public member function	
void push_back (const T&)	Add a new node to the end of a dlist. ‣ Pros : constant time insertion straightforward to implement	O (1)
bool erase (iterator)	Renew the _prev, _next pointers of the nodes that are before and after the unwanted node and release the dynamically-allocated memory of the node. ‣ Pros : constant time deletion straightforward to implement	O (1)
iterator find (const T&)	Linearly search from the first element to the end Return the iterator pointing to the wanted element or return end() if not found. ‣ Pros : Easily to implement	O (N)
void sort () const	Call helper function : void quickSort (iterator, iterator) const.	O (NlogN)
	Auxiliary member function	
void quickSort (iterator, iterator)	1. Get the pivot iterator by calling iterator partition (iterator, iterator) const. 2. Recursively calling void quickSort (iterator, iterator) to sort the sub-dlists.	O (NlogN)
iterator partition (iterator, iterator) const	Rearrange the order of the elements according to their comparison with the pivot, which is the right-most entry of the given segmentation of a dlist. 1. Entries in left-hand side of the pivot are smaller than the pivot, and entries in right-hand side are no smaller than the pivot. 2. Return the iterator pointing to the pivot	O (N)

‣ Comparison of different sorting methods:

# of entries	Quick Sort	Insertion Sort	Ref program (bubble sort)
	O (NlogN)	O (N^2)	O (N^2)
10,000	0.01s	0.19s	1.02
50,000	0.04s	7.79s	26.46
100,000	0.06s	44.08s	101.5s

Even though insertion sort and ref program are both O(N^2) algorithms, numbers of operations of insertion sort are much less than bubble sort. Compared with traditional implementation of insertion sort in a static array, that of insertion sort in doubly-linked list doesn't require right shifts of entries, but insertion of a node in the right position.

III. Binary Search Tree

Binary Search Tree	
Illustration of binary search tree	

Private data member	
BSTreeNode<T>* _root	Pointing to the root of BST.
BSTreeNode<T>* _tail	Pointing to the dummy node(_tail), which is the right-most node.
size_t _size	Numbers of existing nodes
mutable bool _connectT	Whether dummy node(_tail) is connect on the BST

Iterator Class		
operator ++	Operator ++ will make the iterator move to the next node in order, and stop until it points to _tail. **Case1 : Node has right child.** After moving to its right children, iterator moves to the left-most node, which is the next node. **Case2 : Node doesn't have right child.** Iterator will move upward to its parent, until the value of its parent is greater than that of the node.	O (log N)
operator - -	Operator - - will make the iterator move to the previous node in order, and stop until it points to the left-most node of the BST **Case1 : Node has left child.** After moving to its left children, iterator moves to the right-most node, which is the next node. **Case2 : Node doesn't have left child.** Iterator will move upward to its parent, until the value of its parent is no greater than that of the node.	O (log N)

Binary Search Tree		
	Public member function	
void insert (const T&)	1. Before inserting and after inserting, it will call disconnectTail() and reconnectTail() respectively to make codes more elegant. 2. Binary search : If the value of inserted node is smaller than a node, it will move to its left children; otherwise it will move to its right children. 3. When a children pointer is NULL, insert the node to that position.	O (logN)
bool erase (iterator)	Before erasing and after erasing, it will call void disconnectTail() and reconnectTail() respectively to make codes more elegant. **Case1 : Node to erase has no children** Directly release the memory and maintain its parent's child. **Case2 : Node to erase has one child** After maintaining its parent's child and its child's parent, release the memory. **Case3 : Node to ease has two children** Swap the value of the node and its next node. It will reduce to either Case1 or Case 2.	O (logN)
iterator find (const T&)	Call helper function : bool search (BSTreeNode<T>*&, const T&) const) Return the iterator pointing to the wanted element or return end() if not found.	O (logN)
	Auxiliary member function	
bool search (BSTreeNode<T>*&, const T&) const	Binary search from _root node. The argument BSTreeNode<T>*& will be renew to a pointer pointing to the node with target value if the function return true.	O (logN)
bool findMin (BSTreeNode<T>*, BSTreeNode<T>*&) const	1. The first argument BSTreeNode<T>* is the sub-root of the sub-tree. 2. The second argument BSTreeNode<T>*& will be renew to a pointer pointing to the left-most node of the sub-tree.	O (logN)
bool findMax (BSTreeNode<T>*, BSTreeNode<T>*&) const	1. The first argument BSTreeNode<T>* is the sub-root of the sub-tree. 2. The second argument BSTreeNode<T>*& will be renew to a pointer pointing to the right-most node of the sub-tree.	O (logN)
void disconnectTail() const	Remove dummy node(_tail) from a BST. It will make erase and insert more consistent when dealing with diffrenet nodes.	O (logN)
void reconnectTail() const	Reconnect dummy node(_tail) to the right-most node of a BST.	O (logN)

IV. Experiments

‣ Random Insertion : randomly insert 10,000,000 entries

Data Structure	Dynamic Array	Doubly Linked List	Binary Search Tree
Complexity	O(1)	O(1)	O(logN)
Time used	1.78s	0.99s	16.69s
Memory used	896.1 MB	465.1MB	620.1 MB

Time used : Binary Search Tree > Dynamic Array > Doubly Linked List

Since **Binary Search Tree** needs to maintain its data structure in order with O(logN) algorithm. As for **Dynamic Array**, it requires expansion when its size is equal to its capacity, which involves copy entries from one to the other. **Doubly Linked List** only needs to insert a new node after the last entry, which spends less time than Dynamic Array.

‣ Worst Case Insertion : insert “zzzz” - “aaaa” in sorted order (eg. adta -s xxxx)

Data Structure	Dynamic Array	Doubly Linked List	Binary Search Tree
Complexity	O(1)	O(1)	**O(N)**
Time used	13.95s	13.81s	478.1s
Memory used	62.17 MB	51.38 MB	58.43 MB

Time used : Binary Search Tree > Dynamic Array > Doubly Linked List

Since the complexity of Binary Search Tree insertion become O(N) and its data structure is linear, it requires much more time than random insertion. By the way, the height of a tree is N, then complexities of every single operation on BST will become O(N).

‣ Random Deletion : randomly delete 10,000 entries out of 100,000

Data Structure	Dynamic Array	Doubly Linked List	Binary Search Tree
Complexity	O(1)	**O(N)**	**O(N)**
Time used	0.02s	1.39s	27.39s
Memory used	8.172 MB	4.824 MB	6.371 MB

Time used : Binary Search Tree > Doubly Linked List > Dynamic Array

Since random deletion will be executed by calling **_container.erase(getPos(pos))**, where the function : **iterator getPos(size_t)** is O(N) algorithm in **Doubly Linked List** and **Binary Search Tree**, then they spend much more time than **Dynamic Array**, which is executed randomly access with O(1) algorithm. Compared with Doubly Linked List, Binary Search Tree not only requires iterator traversal with O(N), but nodes maintenance and determination of three different cases.