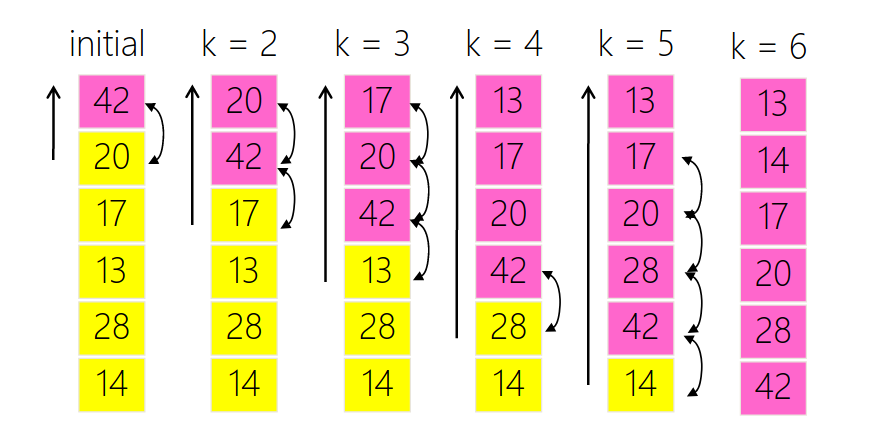
**Programming Assignment #3**

Lecturer: Prof. Jaesik Park  
Teaching Assistants: Minjae Jeong, Suhyeon Jeong, Seunguk Do, Sangjun Lee

|  |
| --- |
| *\*\*\*\* PLEASE READ THIS GRAY BOX CAREFULLY BEFORE STARTING THE ASSIGNMENT \*\*\*\**  Due date: 11:59PM May 12, 2023  Evaluation policy:   * Late submission penalty   + 11:59PM May 12 ~ 11:59PM May 13     - Late submission penalty (30%) will be applied to the total score   + After 11:59PM May 13     - 100% penalty is applied for that submission * Your code will be automatically tested using an evaluation program   + Each problem has the maximum score   + A score will be assigned based on the behavior of the program * Please check if your program generates “submit.txt” same as “answer.txt”, after running ./pa3.exe 0 * We won’t accept any submission via email - it will be ignored * Please do not use the containers in C++ standard template library (STL)   + Such as:     - #include <queue>     - #include <vector>     - #include <stack>   + Any submission using the containers in STL will be disregarded   Any questions?   * Please use PLMS - Q&A board. |

1. Insertion Sort (2 pts)



1. Implement a function that sorts a given array using the **insertion sort** algorithm in ascending order. You can modify sort.cpp and sort.h files for this problem.
2. Input & Output

Input: A sequence of commands

* (‘insert’,integer): insert integer into the array (no need to print current state)
* (‘insertionSort’,NULL): sort the array using the selection sort algorithm (print each sorting step)

Output:

* Every value in the array for each sorting step including the initial step, string separated with the white space (please use built-in function to print the array).
* We won’t test array size over 20 or array size of 0.

1. Example Input & Output

|  |  |
| --- | --- |
| Input | Output |
| [('insert',42),('insert',20),('insert',17),('insert',13),('insert',28),('insert',14),('insertionSort',NULL)] | 42 20 17 13 28 14  20 42 17 13 28 14  17 20 42 13 28 14  13 17 20 42 28 14  13 17 20 28 42 14  13 14 17 20 28 42 |
| [('insert',5),('insert',6),('insert',4),('insert',3),('insert',2),('insert',1),('insertionSort',NULL),('insert',0),('insert',7),('insertionSort',NULL)] | 5 6 4 3 2 1  5 6 4 3 2 1  4 5 6 3 2 1  3 4 5 6 2 1  2 3 4 5 6 1  1 2 3 4 5 6  1 2 3 4 5 6 0 7  0 1 2 3 4 5 6 7  0 1 2 3 4 5 6 7 |

1. Example execution

|  |
| --- |
| >> ./pa3.exe 1 "[('insert',42),('insert',20),('insert',17),('insert',13),('insert',28),('insert',14),('insertionSort',NULL)]"  [Task 1]  42 20 17 13 28 14  20 42 17 13 28 14  17 20 42 13 28 14  13 17 20 42 28 14  13 17 20 28 42 14  13 14 17 20 28 42 |

1. Recursive Merge Sort (2 pts)

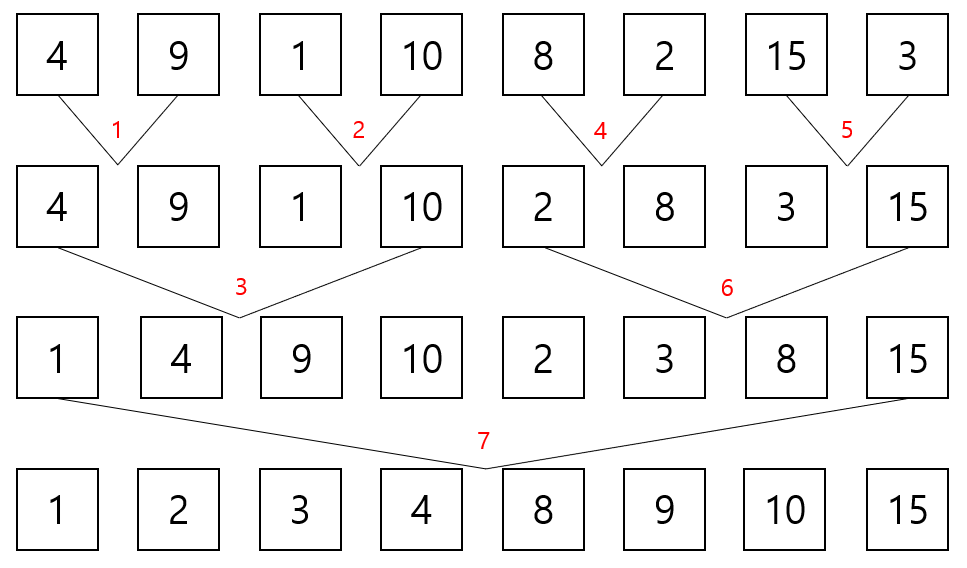


Figure 1. Process of recursive merge sort algorithm.

1. In this merge sort algorithm, we define the term ‘drop’ as the process of selecting an element from a *right-side* sublist during the merging of two sublists. Note that a ‘drop’ occurs at every allocation of elements in a merged sublist, and when elements of a left-side sublist remain.

In Figure 2, two sublists are merged, occurring ***three*** ‘drop’ operations, as highlighted with red arrow.

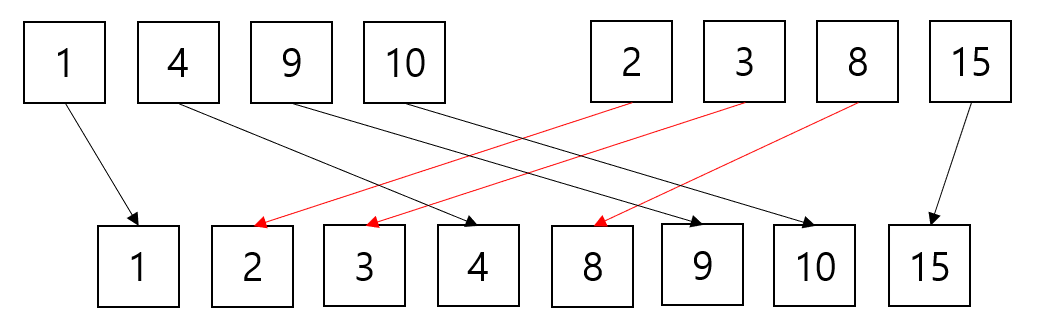


Figure 2. An illustration of comparing and merging two sorted sublists in merge sort algorithm. A red arrow indicates that a drop operation occurred, and three drop operations occur in total.

1. Implement a function that shows process of recursive merge sort algorithms and counts the number of ‘drop’ operations. It will sort a given array in ascending order. You can modify sort.cpp and sort.h files for this problem.
2. Input & Output

Input: A sequence of commands

* (‘insert’,integer): insert integer into the array. integer is unique, and the total number of insert commands is power of 2.
* (‘mergeSort’,NULL): sort the array using the merge sort algorithm, and count ‘drop’ operations. This will be given at the last of the command sequence.

Output:

* Every value in the array for each sorting step including the initial step, string separated with the white space (please use built-in function to print the array).
* The number of ‘drop’ operations.

1. Example Input & Output

|  |  |
| --- | --- |
| Input | Output |
| [('insert',42),('insert',20),('insert',17),('insert',13),('mergeSort',NULL)] | 42 20 17 13  20 42 17 13  20 42 13 17  13 17 20 42  4 |
| [('insert',5),('insert',6),('insert',4),('insert',3),('insert',2),('insert',1),('insert',10),('insert',7),('mergeSort',NULL)] | 5 6 4 3 2 1 10 7  5 6 4 3 2 1 10 7  5 6 3 4 2 1 10 7  3 4 5 6 2 1 10 7  3 4 5 6 1 2 10 7  3 4 5 6 1 2 7 10  3 4 5 6 1 2 7 10  1 2 3 4 5 6 7 10  7 |

1. Example execution

|  |
| --- |
| >> ./pa3.exe 2 "[('insert',5),('insert',6),('insert',4),('insert',3),('insert',2),('insert',1),('insert',10),('insert',7),('mergeSort',NULL)]"  [Task 2]  5 6 4 3 2 1 10 7  5 6 4 3 2 1 10 7  5 6 3 4 2 1 10 7  3 4 5 6 2 1 10 7  3 4 5 6 1 2 10 7  3 4 5 6 1 2 7 10  3 4 5 6 1 2 7 10  1 2 3 4 5 6 7 10  7 |

1. Duplication-Aware BST Insertion / Deletion (4 pts)

DABSTree class is a subclass of BinarySearchTree class. You may use the insert and remove functions of BinarySearchTree, provided in bst.cpp and bst.h.

1. Conventional binary search tree does not permit duplicated element. Now, let’s consider how to deal with duplication problem for the real world. A simple solution is to define another integer variable in an element, ‘count’, recording the number of duplications of a key. For this question, you don’t need to consider value in each element.

For instance, after inserting elements with the keys [12, 5, 17, 8, 5, 16, 17, 5, 20, 8, 20], the tree looks alike the below figure.

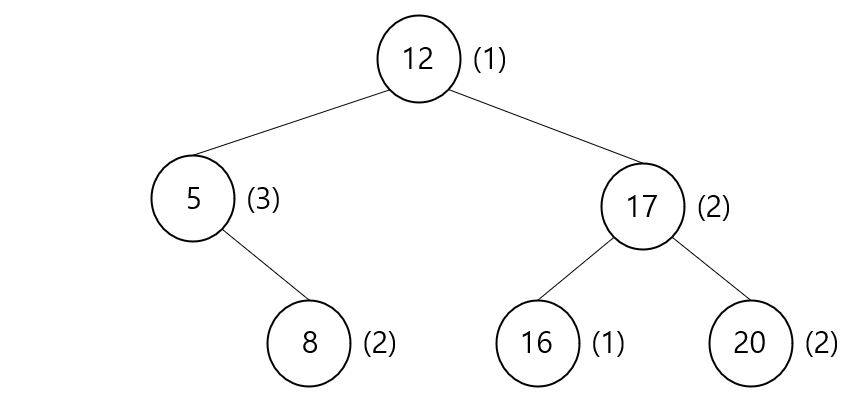


Figure 3. An example of duplication-aware binary search tree.

1. Implement functions that **insert**sand **remove**s an element into a duplication-aware binary search tree. You can modify dabst.cpp and dabst.h files for this problem.
2. Input of DABSTree::insert

Input:

* Key of the element to be inserted. The key has a positive integer value.
* As shown in Figure 3, add 1 to ‘count’ of the element if there is the key already exists in the tree.

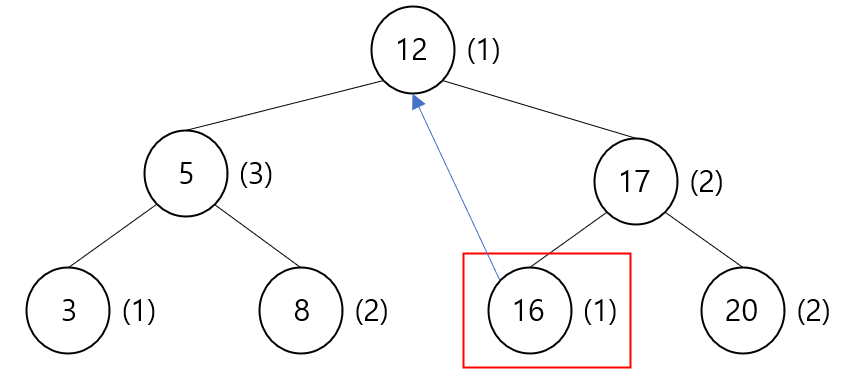


Figure 4. An illustration of deleting an element in duplication-aware binary search tree.

1. Input & output of DABSTree::remove

Input:

* Key of an element to be deleted. Note that deletion means subtracting 1 from ‘count’ of the element.
* If ‘count’ value of the target key is 1, delete the element.

Output:

* If the key does not exist in the tree, print -1 and do not delete any element.
* *Note that replace the smallest key in right subtree when delete an element with degree 2.*

1. task\_3 prints
   1. The results of preorder and inorder traversal of the current constructed tree when each operation is performed successfully.
   2. -1, when DABSTree::remove is called and the key does not exist in the tree.
2. Example Input & Output

|  |  |
| --- | --- |
| Input | Output |
| [('insert',3),('insert',5),('insert',5),('insert',9),('remove',9)] | 3(1)  3(1)  3(1) 5(1)  3(1) 5(1)  3(1) 5(2)  3(1) 5(2)  3(1) 5(2) 9(1)  3(1) 5(2) 9(1)  3(1) 5(2)  3(1) 5(2) |
| [('insert',7),('insert',7),('insert',1),('remove',1),('remove',1),('remove',7)] | 7(1)  7(1)  7(2)  7(2)  7(2) 1(1)  1(1) 7(2)  7(2)  7(2)  -1  7(1)  7(1) |
| [('insert',4),('insert',2),('insert',10),('insert',9),('insert',15),('insert',1),('remove',1),('remove',4),('remove',10),('insert',15),('insert',2)] | 4(1)  4(1)  4(1) 2(1)  2(1) 4(1)  4(1) 2(1) 10(1)  2(1) 4(1) 10(1)  4(1) 2(1) 10(1) 9(1)  2(1) 4(1) 9(1) 10(1)  4(1) 2(1) 10(1) 9(1) 15(1)  2(1) 4(1) 9(1) 10(1) 15(1)  4(1) 2(1) 1(1) 10(1) 9(1) 15(1)  1(1) 2(1) 4(1) 9(1) 10(1) 15(1)  4(1) 2(1) 10(1) 9(1) 15(1)  2(1) 4(1) 9(1) 10(1) 15(1)  9(1) 2(1) 10(1) 15(1)  2(1) 9(1) 10(1) 15(1)  9(1) 2(1) 15(1)  2(1) 9(1) 15(1)  9(1) 2(1) 15(2)  2(1) 9(1) 15(2)  9(1) 2(2) 15(2)  2(2) 9(1) 15(2) |

1. Example execution

|  |
| --- |
| >> ./pa3.exe 3 "[('insert',3),('insert',5),('insert',5),('insert',9),('remove',9)]"  [Task 3]  3(1)  3(1)  3(1) 5(1)  3(1) 5(1)  3(1) 5(2)  3(1) 5(2)  3(1) 5(2) 9(1)  3(1) 5(2) 9(1)  3(1) 5(2)  3(1) 5(2) |

1. AVL Tree Insert / Remove (3 pts)

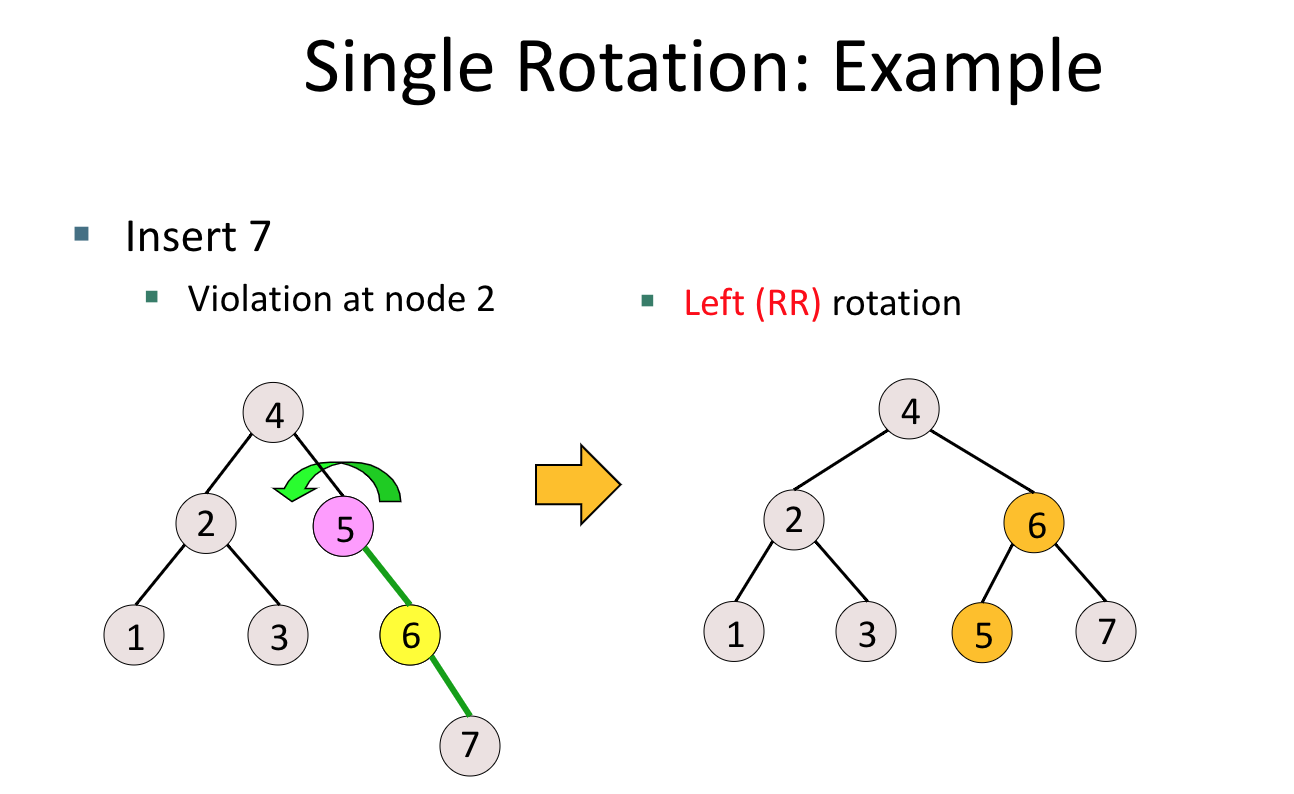


Figure 5. Example of left rotation to resolve RR imbalance

AVLTree class is a subclass of BinarySearchTree class. You may use the insert and remove functions of BinarySearchTree, provided in bst.cpp and bst.h.

1. Implement a function that inserts and deletes an element into a AVL tree. The insertion and deletion might cause the tree to violate its properties (***Imbalance***). Your code should be able to resolve the imbalances (LL, LR, RL, RR) of the tree. You can modify avl.cpp and avl.h files for this problem.
2. Input & Output of AVLTree::insert (insert function for AVL tree)

Input: key of element to be inserted. (keys are given only positive value)

Output:

* -1, if the key already exists in the tree, 0 otherwise.

1. Input & Output of AVLTree::remove (remove function for AVL tree)

Input: key of element to be removed. (keys are given only positive value)

Output:

* -1, if the key does not exist in the tree, 0 otherwise.
* *Note that replace the smallest key in right subtree when remove the element with degree 2 in BST deletion stage.*

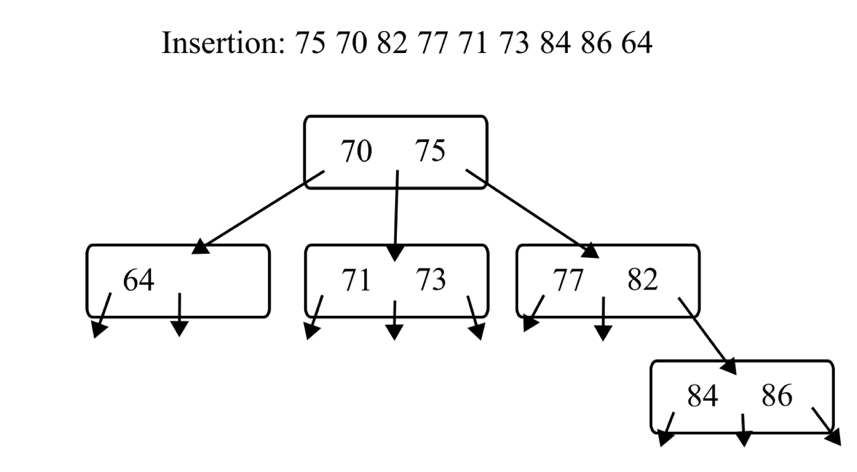
1. task\_4 prints
   1. The results of preorder and inorder traversal of the current constructed tree after insertion/deletion, if insert/remove element has been successed.
   2. -1 , if the key does exist for insertion, or if the key does not exist for deletion.
2. Example Input & Output

|  |  |
| --- | --- |
| Input | Output |
| [('insert',4),('insert',6),('insert',0),('remove',7)] | 4  4  4 6  4 6  4 0 6  0 4 6  -1 |
| [('insert',4),('insert',2),('insert',10),('insert',9),('insert',5),('remove',4),('remove',3),('insert',7)] | 4  4  4 2  2 4  4 2 10  2 4 10  4 2 10 9  2 4 9 10  4 2 9 5 10  2 4 5 9 10  5 2 9 10  2 5 9 10  -1  5 2 9 7 10  2 5 7 9 10 |

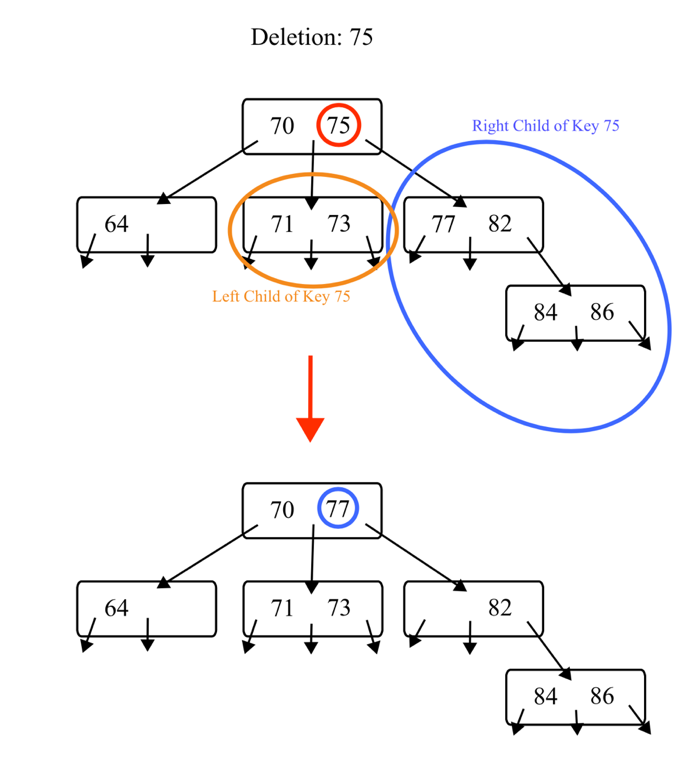
1. Example execution

|  |
| --- |
| >> ./pa3.exe 4 "[('insert',4),('insert',6),('insert',0),('remove',7)]"  [Task 4]  4  4  4 6  4 6  4 0 6  0 4 6  -1 |

1. MST (Multi-way Search Tree) Insertion / Deletion (4 pts)
2. Let’s consider multi-way search tree (MST). When inserting or deleting a key, the property of multi-way search tree should be always satisfied: (1) Every key in a node is ordered increasingly, (2) Every key is greater than all keys stored in its left subtree & less than all keys in its right subtree.

Figure 6. Example of Inserting Keys into an empty MST

When deleting a key from a node, if the key has a right subtree replace it with the smallest key in the right subtree. And if the key has only a left subtree, then replace it with the largest key in the left subtree. If the key has no child, it does not need to be replaced.

****Figure 7. Example of Deleting key ‘75’ from an MST

We define the preorder traversal of node v in MST T as follows: First, visit all the keys in node v in increasing order. Then perform preorder traversal of first child of v in MST T, preorder traversal of second child of v in MST T, and preorder traversal of the last child of v in MST T.

1. Given a multi-way search tree (MST), implement a function that inserts or deletes an element. In this problem **the maximum number of children** for each node of MST is fixed to **three**. You can modify mst.cpp and mst.h files for this problem.

Input & output of MultiWaySearchTree::insert

Input: Key of the element to be inserted. The key has a positive integer value.

Output: Return the -1 if the key already exists in the tree. (If the key already exists, do not insert the element)

Input & output of MultiWaySearchTree::remove

Input: Key of the element to be deleted.

Output: Return the -1 if the key does not exist in the tree. If the key does not exist, do not delete any element.

1. task\_5 prints
   1. the return for each insertion/deletion and
   2. the result of preorder traversal of the constructed tree.
2. Example Input & Output

|  |  |
| --- | --- |
| Input | Output |
| [('insert',4),('insert',6),('insert',6),('insert',7),('remove',7)] | 4  4 6  -1  4 6 7  4 6 |
| [('insert',4),('insert',2),(‘remove’,2),('remove',2),('remove',4),('insert',4)] | 4  2 4  4  -1  4 |
| [('insert',4),('insert',2),('insert',10),('insert',9),('insert',15),('insert',1),('remove',1),('remove',4),('remove',10)] | 4  2 4  2 4 10  2 4 9 10  2 4 9 10 15  2 4 1 9 10 15  2 4 9 10 15  2 9 10 15  2 9 15 |
| [('insert',75),('insert',70),('insert',82),('insert',77),('insert',71),('insert',73),('insert',84),('insert',86),('insert',64),('remove',75)] | 75  70 75  70 75 82  70 75 77 82  70 75 71 77 82  70 75 71 73 77 82  70 75 71 73 77 82 84  70 75 71 73 77 82 84 86  70 75 64 71 73 77 82 84 86  70 77 64 71 73 82 84 86 |
| [('insert',75),('insert',70),('insert',82),('insert',77),('insert',71),('insert',73),('insert',84),('insert',86),('insert',64),('insert',80),('remove',75)] | 75  70 75  70 75 82  70 75 77 82  70 75 71 77 82  70 75 71 73 77 82  70 75 71 73 77 82 84  70 75 71 73 77 82 84 86  70 75 64 71 73 77 82 84 86  70 75 64 71 73 77 82 80 84 86  70 77 64 71 73 80 82 84 86 |

1. Example execution

|  |
| --- |
| >> ./pa3.exe 5 "[('insert',4),('insert',6),('insert',6),('insert',7),('remove',7)]"  [Task 5]  4  4 6  -1  4 6 7  4 6 |

1. B-tree Insertion / Deletion (Bonus 3 pts)
2. Given a B-tree, implement a function that inserts or deletes an element. This function inserts a key if there is no same key value in the B-tree. If the key already exists in the B-tree, you need to delete an element having that key.   
   Your B-tree should maintain the B-tree property even after the insertion or deletion.   
   The order of the B-tree is positive integer larger than 2. For the even order case, use the key at the ⌊M/2⌋ index of the node as an median when overflow occurs. (See the first case of the Example Input & Output)  
   After finishing every insertion/deletion process, print the constructed B-tree through postorder traversal method. If there is no element in the B-tree, print -1. You can modify b\_tree.cpp and b\_tree.h files for this problem.
3. Input & output

Input: Commands (The order 'M', and multiple 'key's)

* ('M',integer): specify the order of a B-tree.
* ('key',positive integer): a key to be inserted or deleted.

Output: The results of postorder traversal of the constructed B-tree.

1. Example Input & Output

|  |  |
| --- | --- |
| Input | Output |
| [('M',4),(‘key’,1),(‘key’,2),(‘key’,3),(‘key’,4)] | [1] [3, 4] [2] |
| [('M',6),('key',9),(‘key’,1),(‘key’,7), (‘key’,2),(‘key’,3),(‘key’,6),(‘key’,31), (‘key’,49),(‘key’,3)] | [1, 2] [7, 9, 31, 49] [6] |
| [('M',3),('key',25),('key',38),('key',40),('key',20),('key',30),('key',35),('key',20)] | [25] [35] [40] [30, 38] |
| [('M',4),('key',19),('key',32),('key',4),('key',12),('key',12),('key',7),(‘key’,1), (‘key’,2),(‘key’,3),(‘key’,5),(‘key’,12),(‘key’,6),(‘key’,8),(‘key’,9)] | [1] [3] [2] [5] [7] [9, 12] [32] [6, 8, 19] [4] |
| [('M',3),('key',19),('key',8),('key',2),('key',15),('key',6),('key',13),('key',7)] | [2] [7] [6] [13] [19] [15] [8] |

1. Example execution

|  |
| --- |
| >> ./pa3.exe 6 “[('M',4),(‘key’,1),(‘key’,2),(‘key’,3),(‘key’,4)]”  [Task 6]  [1] [3, 4] [2] |