***Task 1***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Unsorted,singly linked | Sorted,singly linked | Unsorted,doubly linked | Sorted,doubly linked |
| Search(L,k) | Linear (we need to find from n elements) | Linear (we need to find from n elements) | Linear (we need to find from n elements) | Linear (we need to find from n elements) |
| Insert(L,x) | Constant(as list is unsorted we will insert from the beginning) | Linear(at first we need to find the place for inserting our element in the correct place, than we will insert it, if we assume that search isn’t a part of the insert, we could say that we need just constant time) | Constant(as list is unsorted we will insert from the beginning) | Linear(at first we need to find the place for inserting our element in the correct place, than we will insert it, if we assume that search isn’t a part of the insert, we could say that we need just constant time) |
| Delete(L,x) | Linear ( we need to find the previous element of our x and then delete) | Linear ( we need to find the previous element of our x and then delete) | Constant(we have previous element here) | Constant(we have previous element here) |
| Successor(L,x) | Linear (we need to find it) | Constant(successor is the next element of our x) | Linear (we need to search it) | Constant(successor is the next element of our x) |
| Predecessor(L,x) | Linear (we need to find it) | Linear (we need to reach to the previous element of x) | Linear (we need to find it) | Constant (predecessor is the previous element and we have pointer to the previous) |
| Minimum(L) | Linear(we need to find it) | Constant (minimum will be the first element) | Linear (we need to find it) | Constant (minimum will be the first element) |
| Maximum(L) | Linear (we need to find it) | Linear (we need to reach to the tail) | Linear (we need to find it) | linear Linear (we need to reach to the tail, but if we have tail we can make the algorithm constant) |

***Task 2***

1. TREE-DELETE(T, z)
2. if z.left == NULL
3. TRANSPLANT(T, z, z.right)
4. elseif z.right == NULL
5. TRANSPLANT(T, z, z.left)
6. else
7. y = TREE-MAXIMUM(z.left)
8. if y.p != z
9. TRANSPLANT(T, y, y.left)
10. y.left = z.left
11. y.left.p = y
12. TRANSPLANT(T, z, y)
13. y.right = z.right
14. y.right.p = y

For implementing the strategy, we could randomly decide what will use TREE-DELETE successor or predecessor.

***Task 3***

C is wrong because we have node with value 911 and for searching 363 we will go to the left child of 911. After that we see 912 which cannot be in the left subtree of 911.

E is wrong because we have node with value 347, 363>347 so we will go to the right subtree of 347, but there is a node with value 299.