Problem 1 (4+1 points). You are given two lists, L and P. List L contains generic objects. List P contains non-negative integers sorted in ascending order. The operation **printLots**(L, P) will print the elements in L that are in positions specified by P. For instance, if $P = [1, 3, 4, 6]$, the elements in positions 1, 3, 4, and 6 in L are printed.

(1) Write the procedure **printLots**(L, P) using only the basic list operations.

(2) What is the running time of your procedure? You must show your work (i.e., justify your answer).

a. Worst case running time is O(n^2). This is because of the nested while loops running for a variable amount of times that depends on size of n of both lists. Outside of the while loops each statement is O(1)

```
b.  void printLots(Linkedlist* L, Linkedlist* P)
c.  {
d.      Node* LTemp = L->head;
e.      Node* PTemp = P->head;
f.      int indexOfL = 0;
g.      while (PTemp != nullptr)
h.      {
i.          while(indexOfL < PTemp->data)
j.          {
k.              LTemp = LTemp->next;
l.              indexOfL++;
m.          }
n.          cout << LTemp->data << " ";
o.          PTemp = PTemp->next;
p.      }
q.      cout << endl;
r.  }
```

Problem 2 (4 points). Write an algorithm for printing a **singly** linked list in reverse, using only **constant extra space** and **linear** time. This instruction implies that you cannot use recursion.

```
void Linkedlist::reverse()
{
   Node* curr = this->head;
```

```
    Node* newHead = nullptr;
    Node* temp = nullptr;

    while (curr != nullptr)
    {
        temp = curr->next;
        curr->next = newHead;
        newHead = curr;
        curr = temp;
    }
    this->head = newHead;
}
```

Problem 3 (4+1 points). Given two **sorted** lists, L_1 and L_2, the operation **intersection**(L_1, L_2) will compute $L_1 \cap L_2$ (the intersection of L_1 and L_2: the set of elements which are in both lists). For example, if $L_1 = [0, 2, 4, 6]$ and $L_2 = [1, 2, 3, 6, 7]$, the list $[2, 6]$ is returned.

(1) Write the procedure **intersection**(L_1, L_2) using only basic list operations.

(2) What is the running time of your procedure? You must show your work (i.e., justify your answer).

a. The worst case runtime of this procedure is O(n) as we only have one loop that loops for size n or size m of the two lists L1 and L2 respectively. If any of the lists are null or empty then the runtime would be O(1) which is the best case. Each statement outside of the loop runs O(1).

b.

```
c.  void Intersection(Linkedlist* L1, Linkedlist* L2)
d.  {
e.      Node* L1_pos = L1->head;
f.      Node* L2_pos = L2->head;
g.
h.
i.      while(L1_pos != NULL && L2_pos != NULL)
j.      {
k.          if(L1_pos->data == L2_pos->data)
l.          {
m.
n.              cout << L1_pos->data << " ";
```

```
            L1_pos = L1_pos->next;

            L2_pos = L2_pos->next;

        }

        else if(L1_pos->data < L2_pos->data)

        {

            L1_pos = L1_pos->next;

        }

        else if(L1_pos->data > L2_pos->data)

        {

            L2_pos = L2_pos->next;

        }

    }

  cout << endl;

}
```

Problem 4 (4 points). Suppose we have a pointer to a node in a **singly** linked list that is guaranteed to not be the last node in the list. We don't have pointers to any other nodes (except by following links). Describe an $O(1)$ algorithm that logically removes the value stored in such a node from the linked list.
Hint. Involve the next node.

To solve this in O(1) is pretty simple. You just have to copy the data and pointer of the next node into the given node. So essentially replace your given node by copying the next node into your given node. Doing this gets rid of the data of your current node and skips the original next node since you updated the pointer of your current node to the next. So now your node's next pointer points to what was originally the next next pointer.
In pseudo code this would look like if given is the given node pointer:

```
given.data = given.next.data;
given.data = given.next.next;
```

So essentially you have copied the next node into the current one and by updating the next pointer you skip over that original next node so you do not have any duplicates.

Rubric:
Problem 1
Part (1):

(i) 0 pts

- not attempted or illegible.

(ii) 1 pt
- Fragmentary. Clear misunderstanding.
- Insubstantial attempt made.

(iii) 2 pts
- Algorithm is incorrect.
- Algorithm relies on non-standard or "advanced" list operations, e.g., merge.

(iv) 3 pts
- Algorithm is correct.
- Algorithm uses only basic list operations.

(v) 4 pts
- Algorithm is correct.
- Algorithm uses only basic list operations.
- Algorithm runs in $O(|L| + |P|)$ time.

Part (2):

(i) 0 pts
- not attempted or illegible.

(ii) 0.4 pts
- Attempted.

(iii) 0.6 pts
- Details are not given.
- Final answer is correct but is a vast over/underestimate. (E.g, the running time is $O(n)$ but $O(n^2)$ is give.)

(iv) 1 pt
- Analysis clearly articulates the justification for the running time. Details are given.

Problem 2

(i) 0 pts
- not attempted or illegible.

(ii) 1 pt
- Fragmentary. Clear misunderstanding.
- Insubstantial attempt made.

(iii) 2 pts
- Algorithm is incorrect.
- Algorithm relies on non-standard or "advanced" list operations, e.g., merge.

(iv) 3 pts
- Algorithm is correct.
- Algorithm uses only basic list operations.

(v) 4 pts
- Algorithm is correct.
- Algorithm uses only basic list operations.
- Algorithm uses only **constant extra space** and **linear** time.

Problem 3

Part (1):

(i) 0 pts
- not attempted or illegible.

(ii) 1 pt
- Fragmentary. Clear misunderstanding.
- Insubstantial attempt made.

(iii) 2 pts
- Algorithm is incorrect.
- Algorithm relies on non-standard or "advanced" list operations, e.g., merge.

(iv) 3 pts
- Algorithm is correct.
- Algorithm uses only basic list operations.

(v) 4 pts
- Algorithm is correct.
- Algorithm uses only basic list operations.
- Algorithm runs in $O(|L_1| + |L_2|)$ time.

Part (2):

(i) 0 pts
- not attempted or illegible.

(ii) 0.4 pts
- Attempted.

(iii) 0.6 pts
- Details are not given.
- Final answer is correct but is a vast over/underestimate. (E.g, the running time is $O(n)$ but $O(n^2)$ is give.)

(iv) 1 pt
- Analysis clearly articulates the justification for the running time. Details are given.

Problem 4:

(i) 0 pts
- not attempted or illegible.

(ii) 1 pt
- Fragmentary. Clear misunderstanding.
- Insubstantial attempt made.

(iii) 2 pts
- Algorithm is incorrect.
- Algorithm relies on non-standard or "advanced" list operations, e.g., merge.

(iv) 3 pts
- Algorithm is correct.
- Algorithm uses only basic list operations.

(v) 4 pts

- Algorithm is correct.
- Algorithm uses only basic list operations.
- Algorithm runs in $O(1)$ time.