Assignment 2

Due 23:59, Friday, October 4, 2019

CS ID 1:

CS ID 2:

Instructions:

1. Do not change the problem statements we are giving you. Simply add your solutions by editing this latex document.
2. Take as much space as you need for each problem. You'll tell us where your solutions are when you submit your paper to gradescope.
3. Export the completed assignment as a PDF file for upload to gradescope.
4. On gradescope, upload only **one** copy per partnership. (Instructions for uploading to gradescope will be posted on the HW2 page of the course website.)

1. **Dead Heads (25 points).**

   In some cases, it's easier to handle the special case of deleting the head of a linked-list or inserting a new node in front of the head of a linked-list by using a *dead head* (sometimes called a *dummy head*, or *sentinel*). The dead head is always the first node in the linked-list but it contains no data. (You can think of it as an empty node!) Its `next` pointer points to the "real" head of the linked-list, that is the first node in the linked-list that contains valid data. So an empty linked-list is represented by a dead head whose `next` pointer is `NULL`. The user specifies the linked-list by using a pointer to the list's dead head. For this problem, you will fix some implementations that use dead-headed linked-lists as a private member representation of a `List` class.

   Here is the `List` class definition that we will examine and expand:

```
class List{
  public:
    void addAt(int pos, int elt);
    void delAll(int x);
  private:
    struct Node{
      int data;
      Node * next;
      Node(int elt = 0,Node * p = NULL):data(elt),next(p){}
    };
    Node * head;
    Node * walk(Node * curr, int k);
    void delNode(Node * p);
};

List::Node * List::walk(List::Node * curr, int k){
  // walks curr forward k steps
  if (k<=0 || curr == NULL)
    return curr;
  else return walk(curr->next, k-1);
}
```

   (a) (2 points) Write the no-argument constructor for the above `List` class if the class is designed so that the empty list is represented by a single dead head node.

```
List::List(){
    head = new Node();
}
```

   (b) (2 points) Suppose that you have designed class `List` to contain a linked list implemented *without* a dead head. (We will refer to this implementation as the "Live Head" implementation, which is not an official term!) Write member function `addAt(int pos, int e)` which creates a new node at position `pos` whose data is `e`. Note that in this case we consider the first data element of the list to be position 0, and that in an empty list, `head == NULL`. Please use the given `walk` function to advance a pointer along the list.

```
void List::addAt(int pos, int elt){
  if (pos <= 0)
    head = new Node(elt,head);
  else {
    Node * t = walk(head,pos - 1);
    t->next = new Node(elt,t->next);
  }
}
```

(c) (2 points) Rewrite the `addAt` member function for a list designed to use a dead head at the front of the list. Note that in this case we consider the first data element of the list to be position 0. You may assume that member variable `head` is not `NULL`, even if the list is empty.

```
void List::addAt(int pos, int elt){

    Node * t = walk(head,pos);
    t->next = new Node(elt,t->next);

}
```

(d) (2 points) Research the meaning of the term *cyclomatic complexity*, and use your understanding to answer the following questions.

Which implementation of `addAt` has lower cyclomatic complexity?
● Dead Head ○ Live Head

Given their relative cyclomatic complexities, we expect the number of test cases for the dead head implementation to be ○ greater than ● less than ○ equal to the number we would need for the live head implementation of `addAt`.

(e) (4 points) The following `List` class member function is intended to delete all nodes with data equal to x from a dead-headed linked-list whose last node has a `NULL` next pointer. It doesn't always work. Two lines of code need to be corrected. Write the correct lines in the boxes next to the two lines that are incorrect.

`void List::delAll(int x){`	
`  a = head;`	`Node *a = head;`
`  while ( a->next != NULL ) {`	
`    if ( a->next->data == x ) {`	
`      Node *t=a;`	`Node *t=a->next;`
`      a->next = a->next->next;`	
`      delete t;`	
`    }`	
`    a = a->next;`	`else a = a->next;`
`  }`	
`}`	

In this part of the problem, we will explore an algorithm for removing a node from a singly linked list containing n nodes *given a pointer to the node we wish to remove.*

(f) (3 points) This removal task would be simple if we had a pointer to the node *before* the one we wish to remove. We can be sure there *is* a node before the one we wish to remove if we implement the list with a dead head. Complete the code below to implement the `List` class member function that removes node `p` from the list whose `head` is a dead head, using the comments to guide your algorithm. You may assume that `p` is a valid data element from the linked list.

```
void List::delNode(Node *p){
  Node * t = head;
  /* advance t until it's in a useful place */

  while(t->next != p ) t=t->next;

  /* adjust pointers to remove appropriate node */

  t->next = p->next;

  /* free memory associated with p */

  delete p;

}
```

(g) (1 point) The worst case running time of `delNode` is ○ $\Theta(1)$ ○ $\Theta(\log n)$ ● $\Theta(n)$

(h) (4 points) If we are clever, we can solve the problem of removing a node, given a pointer to it, *without* iterating over the list. To solve this problem, consider the pointer you are given, and ask yourself which node *would* be easy to remove. Is there a way to remove that node, while at the same time preserving its data? The following function

is intended to effectively delete the node pointed to by `p` from a linked-list with a dead head. Complete the code below so that it takes constant time to do this operation. You may assume that `p` points to a valid data node (not the dead head).

```
void List::delNode(Node *p){
  /* set up a pointer to a node that would be easy to remove */

  Node * t = p->next;

  /* preserve the data contained in that node */

  p->data = t->data;

  /* fix pointers */

  p->next = t->next;

  /* free appropriate memory */

  delete t;

}
```

(i) (2 points) You probably noticed that this algorithm fails if the input parameter `p` points to a particular node.

i. The code fails if `p` is a pointer to what node?

```
The tail of the linked-list.
```

ii. Describe how you would change the design of the linked list so that the algorithm above would work for any valid `p` (where "valid" means that it points to a node with data).

```
Add a dummy-tail that contains blank data
and is guaranteed not to be pointed to by p.
```

(j) (3 points) This mechanism for removing a node from a list is commonly referred to as a constant time "hack" because it exhibits some important disadvantages. Choose the applicable disadvantages from the list below.

- ● If `p->data` is a large object, the execution of the function may be very slow.
- ○ The compiler cannot optimize the amount of memory needed by the `Node`.
- ● In order to work, the assignment operator must be implemented for data's type.
- ○ There could be be a memory leak.
- ○ The implementation requires a dead head.
- ○ The constant time code will only work on lists with more than one data element.
- ● An iterator over the list may be invalidated.
- ○ This code may result in a segmentation fault due to dereferencing a `NULL` pointer.

2. **Structured Sorting (22 points).**

We want to sort a sequence of numbers using a stack. We imagine the numbers arrive at the stack in their input order. For example, if the input is 5,1,4,2,3 then 5 arrives first and 3 arrives last. When a number arrives, we must push it onto the stack and then we can perform any number of pop operations (including none). Each pop operation not only pops the top number off of the stack it also outputs the number. Pops on an empty stack are invalid. We want the sequence of numbers that are output to be the original input sequence in non-decreasing order.

For example, the following operation sequence sorts 5,1,4,2,3: $IIOIIOIOOO$ where I means "push In" and O means "pop Out".

(a) (2 points) Some operation sequences attempt to pop from an empty stack or end before the stack is empty. These are called *invalid* sequences. Describe two simple conditions that together ensure that an operation sequence is valid.

For all prefixes of the sequence, the number of I's is at least the number of O's.

The number of I's in the whole sequence equals the number of O's.

(b) (2 points) What input sequence using the numbers 1,2,3 cannot be sorted in this way; meaning no sequence of operations will sort the given input.

2 3 1

(c) (8 points) Prove that it is impossible to sort an input sequence in which the number b is before the number c which is before the number a and $a < b < c$.

Since b arrives before a (and a should be output first of a, b, and c), every sequence that sorts must push b on the stack. The same is true for c when c arrives. However, this places c above b in the stack. Thus c is output (popped) before b, which fails to sort the sequence.

Suppose we add the choice of a queue to our sorting algorithm. Now we can use the operations I, O, E, and D, where I and O push and pop the stack, while E and D enqueue and dequeue

the queue (dequeue also outputs the number that is dequeued). For example, we can sort 5,1,4,2,3 using: $IEIEEDDDOO$.

(d) (2 points) What input sequence using the numbers 1,2,3,4 is sorted by the sequence of operations $EIEIOODD$?

4 2 3 1

(e) (4 points) Is it possible to sort all input sequences using such a stack and queue? If yes then explain how. If no, give a permutation that cannot be output.

No. 4 5 2 3 1 cannot be sorted.

(f) (4 points) Suppose we add two new operations.
S : pop a number from the stack (without output) and enqueue it into the queue; and
Q : dequeue a number from the queue (without output) and push it onto the stack.
Is it possible to sort all input sequences using the operations I, O, E, D, S, and Q? If yes then explain how. If no, give a permutation that cannot be output.

Yes. 1) Enqueue the entire sequence. 2) Repeat the two operations QS to cycle the contents of the queue until the minimum is at the front. 3) Use D to dequeue and output the minimum. 4) Repeat from step 2 until done.

3. **Loop Invariants (18 points).**

Consider the following code for finding a maximum element in an array:

```
// Find a maximum element in the array A.
int findMax( int *A, int n ) {
  return findMaxHelper(A, 0, n);
}

// Return the maximum element in A[left..right-1]
int findMaxHelper(int *A, int left, int right) {
  if( left == right - 1 ) return A[left];
  else {
    int max1 = findMaxHelper(A, left, (right + left) / 2);
    int max2 = findMaxHelper(A, (right + left) / 2, right);
    if max1 > max2 return max1 else return max2;
  }
}
```

i. (5 points) Let n be the length of the array `A` and assume $n \geq 1$. Use induction to prove that this code correctly returns the maximum element in the array.

Claim: `findMaxHelper(A, left, right)` returns the maximum element in `A[left..right-1]`.

Proof: (by induction on $n = \texttt{right} - \texttt{left}$)
Base case: $n = 1$ returns `A[left]` which is correct.
Induction hypothesis: Suppose the claim is true for all $n < k$ (where $k > 1$).
Inductive step: (Prove the claim for $n = k$.)
Let $m = (\texttt{right} + \texttt{left})/2$. Note that $m - \texttt{left} < k$ and $\texttt{right} - m < k$. By the induction hypothesis, `max1` is the maximum of `A[left..m-1]` and `max2` is the maximum of `A[m..right-1]`. The function returns the maximum of `max1` and `max2`, which is the maximum of `A[left..right-1]`. □

ii. (5 points) Let $T(n)$ be the run time of this algorithm. Express $T(n)$ as a recurrence relation. Use repeated substitution (ignore floors/ceilings when calculating $n/2$) to determine a closed-form solution for $T(n)$, i.e., express $T(n)$ as a non-recursive function of n with no summations.

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + 2T(n/2) & \text{if } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &= 1 + 2T(n/2) \\ &= 1 + 2(1 + 2T(n/4)) \\ &= 1 + 2 + 4T(n/4) \\ &= 1 + 2 + 4 + 8T(n/8) \\ &\vdots \\ &= \sum_{i=0}^{\lg n} 2^i = 2n - 1 \end{aligned}$$

(a) (8 points) The following code finds the length of the longest subsequence of strictly increasing numbers in a given array:

```
// Returns the length of the longest subsequence of strictly increasing
// elements in the array A.  A subsequence of array A is not necessarily
// contiguous.
int LISlength(int *A, int n) {
  int *LISendAt = new int[n];
  int best = 0;
  for( int i=0; i<n; i++ ) {
    int r = 1;
    for( int j=0; j<i; j++ ) {
      if( A[j] < A[i] && r < LISendAt[j] + 1 )
        r = LISendAt[j] + 1;
    }
    LISendAt[i] = r;
    if( best < r ) best = r;
  }
  delete[] LISendAt;
  return best;
}
```

For example, for the input list `[11, 2, 5, 3, 6, 1, 8, 9, 2, 7, 10, 22, 12, 13, 14, 1]`, this code should return 9, which is the length of the longest increasing subsequence, `[2, 5, 6, 8, 9, 10, 12, 13, 14]`, in the list. In this problem, you'll use loop invariants and induction to prove that this function operates correctly.

Assume that just before the inner loop (indexed by `j`) begins executing, `LISendAt[k]` is the length of the longest increasing subsequence that ends at entry `k` of `A`, for all `k` where $0 \leq$ `k` $<$ `i`. The loop invariant for the inner loop is then: At the start of the `j`th iteration of the loop, `r` holds the length of the longest increasing subsequence that ends with `A[i]` in `A[0..j-1] A[i]`, where `A[0..j-1] A[i]` is the subsequence of `A` consisting of `j` + 1 elements `A[0]` through `A[j-1]` followed by `A[i]`. Use induction to prove that this loop invariant holds.

Claim: r is the length of the longest increasing subsequence (LIS) in $A[0 \ldots j-1]A[i]$ that ends with $A[i]$.

Proof: (by induction on j)
Base case: (prior to iteration $j = 0$)
$r = 1$, $j = 0$, $A[0 \ldots j-1]A[i] = A[i]$ with length of LIS equal to 1.
Induction hypothesis: True for every iteration $j < m$ (where $m > 0$).
Inductive step: (Show true prior to iteration $j = m$.)
By the induction hypothesis, at the start of iteration $j-1$, r is the length of the LIS in $A[0 \ldots j-1]A[i]$ that ends with $A[i]$. The LIS in $A[0 \ldots j]A[i]$ either ends with $A[j]A[i]$ or it is a LIS in $A[0 \ldots j-1]A[i]$ and has length r (by the induction hypothesis). The first possibility occurs only if $A[j] < A[i]$ and the LIS ending at $A[j]$ plus $A[i]$ (which has length `LISendAt[j] + 1` by our assumption that `LISendAt[k]` is correct for all $0 \leq k < i$ (in particular, for $k = j$)) is longer than r. This is what the if-statement checks in order to update r. □

Blank sheet for extra work.