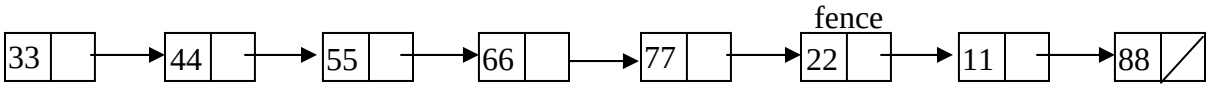
Data Structures and Algorithms: Quiz No. 1

Solutions

1. Suppose we have the following linked list.



To make the fence point to the node whose element's value is 77, it would take these many steps (step=moving the fence from one node to another or assigning it to another pointer).

The fence must be first assigned to the head node (1 Step)
Then the fence must be advanced to the node with value 33, then 44, then 55, then 66, then 77 (5 steps)
Therefore, the total number of steps is 6

2. The constructor of a singly linked list node takes a reference of the node's value and a pointer to the next node as parameters (in this order). The following C++ statement(s) is(are) part of few statements that insert a node in a singly linked list.

Answer d) is the correct one. Remember, we never change the value of the fence pointer when we insert or remove. Hence, we exclude answers a) and c). Answer b) is wrong because, you never include the '&' symbol after a variable's name in a function call or any statement. It is only used in function prototypes and declarations to denote a reference.

a) `fence = new Link<Elem> (item, fence->next);`
b) `fence->next = new Link<Elem> (item &, fence->next);`
c) `fence = new Link<Elem> (item&, fence->next);`
d) `fence->next = new Link<Elem> (item, fence->next);`

3. Suppose we have a pointer Orphan that points to a Link node that is not part of any list. The following C++ statement(s) insert this node in a singly linked list.

Answers a), b), and c) are wrong because they modify the value of the fence pointer. Answer e) is wrong because the fence will be connected to the node on the right side of the new node. This leaves the new node `Orphan` not connected to any node on the left side.

a) `fence = fence->next;Orphan->next=fence->next;`
b) `Orphan->next=fence->next; fence = fence->next;`
c) `Orphan->next=fence->next; fence = Orphan;`
d) `Orphan->next=fence->next; fence->next = Orphan;`

e) `Orphan->next=fence->next; fence->next = Orphan->next;`

4. ADT
a) Stands for templates
b) Stands for abstract classes
c) Stands for any data type
d) Stands for the collection (namespace) of all data types
e) All of the above
An ADT stands for Abstract Data Type, which simply means any data type (answer c))

5. A linked list is most like a:
a) sequence b) set c) chain d) a and b e) a and c f) b and c
In a linked list, order matters and hence, it is similar to a sequence and a chain. In a set, order is not important. Answer e) is the correct one.

6. A linked list of 7 nodes, when compared to an array-based list with 7 elements and MaxSize of 10 elements, requires:
a) More space b) Less space c) Same space
d) Depends on the type of the elements, e.g., integers, double, etc.

The element in a list node (array or linked) can be of any type (character is 2 bytes, integer 4 bytes, double is 8 bytes, a record may be a 100 or more bytes). The size of the pointer on the other hand, in a linked list is always of fixed size, for example, 8 bytes. Now let's say we are storing characters in the nodes of the list and assuming a pointer occupies 8 bytes, then the linked list would occupy (7+1)×(2+8)=80 bytes (we added 1 to 7 because of the head node). The array does not have pointers and has a MaxSize of 10 and therefore it occupies 10×2=20 bytes, which is much less than what the linked list occupied.
Now suppose, we are storing records in the nodes and each record occupies 150 bytes. Then the linked list would consume (7+1)×(150+8)=1264 bytes. The array version on the other hand will take 10×150=1500 bytes, which is in this case more than the linked list. Therefore answer d) is the right one.

7. When a pointer requires 4 bytes and a data element requires 4 bytes, the singly-linked list implementation requires less space than the array-based list implementation when the array would be:
a) less than 1/4 full b) less than 1/3 full c) less than 2/3 full
d) less than 1/2 full e) less than 3/4 full f) never.

If N is the number of nodes in a linked list and it is the number of filled nodes in the array list, then the size of the linked list is N×(4+4)=8N bytes. Now suppose the Array has a max size of M, then the size of the array is 4×M=4M.
The linked list consumes less space when 8N<4M or N<½M. Therefore, answer d) is the right one.

8. The following C++ statement(s) are part of the code that removes a node from a linked list.

a)
```
Link<Elem>* temp=fence->next; fence->next=fence->next->next;
if (tail==fence->next) tail=fence; delete temp;
```

b)
```
Link<Elem>* temp=fence->next; fence->next=fence->next->next;
if (tail==temp) tail=fence; delete temp;
```

c)
```
Link<Elem>* temp=fence->next; fence->next=temp->next;
if (tail==fence->next) tail=fence; delete temp;
```

d)
```
Link<Elem>* temp=fence->next; fence->next=fence->next->next;
if (tail==fence) tail=fence; delete temp;
```

Answer d) is wrong because if `tail` was equal to `fence`, then there would have been nothing to the right of the `fence` to remove.
Answers a) and c) are wrong because we're checking against `fence->next`, which has changed on the previous line. Note that a) and c) are equivalent.
Answer b) is the right one.

9. Here is a series of C++ statements using the list member functions.

```
L1.append(10);  L1.append(20);  L1.append(15);
```

If these statements are applied to an empty list, the result will look like:

a) `<10 20 15>` b) `<10 20 15 |>` c) `<| 10 20 15>`
d) `<15 20 10>` e) `<| 15 20 10>` f) `<15 20 10 |>`

The `|` symbol denotes the fence and initially, we have the following list: `<| >`. So when we append 10, 20, and then 15, they all go to the right side of the fence in the order in which they were appended. Hence, c) is the right answer.

10. By comparing the array-based and linked implementations, the array-based implementation:
a) has slower direct access to elements but faster insert/delete from the current position.
b) has both faster direct access to elements and faster insert/delete from the current position.
c) has faster direct access to elements but slower insert/delete from the current position.
d) has both slower direct access to elements and slower insert/delete from the current position.

Answer c) is the correct one. We can access any element within the array simply by using its index (e.g., A[11], retrieves the 12th element). When we insert into an array-list, we would have to shift all element on the right of the fence to the right. Inversely, when we delete, we have to shift elements to the right. Therefore, the insert and delete operations are slow when it comes to arrays.

11. Suppose we have the list `<15 4 | 20 55 10>` and then execute the following C++ calls

```
e=0;  L1.remove(e);  L1.append(e);  L1.prev();
L1.insert(e);
```

The result will be:
a) `<| 4 15 20 55 10 4>` b) `<15 | 4 20 55 10 20>` c) `<15 | 20 4 55 10 20>`
d) `<15 4 | 20 55 10 20>` e) `<15 | 20 4 55 20 10>` f) `<15 | 0 4 55 10 0>`

The `remove` function returns to us the value to the right of the fence: `<15 4 | 55 10>`. `append`, then inserts this value (20) at the end of the list: `<15 4 | 55 10 20>`. `prev` moves the fence backward: `<15 | 4 55 10 20>` and `insert` inserts this same value (20) after the fence:
`<15 | 20 4 55 10 20>`. It follows that c) is the right answer.

12. For linked-lists (all operations apply directly to the right of the fence), a header node is used:
a) Because there is no other way for the fence pointer to indicate the first element in the list.
b) Because the insert and delete routines won't work correctly in all cases without it.
c) a and b

Answer b) is the right one because if there wasn't this special node, there is no way to insert a node at the start of the list nor delete the first node. a) is not true because we can state that the current element is the one that the fence points to. Hence, if the fence is at the start of the list, we can say that the fence points to the first element.

13. For a list of length **n**, the linked-list's `prev` function requires worst-case time (below, the symbol *O* denotes "in the order of". Example: *O* (4) means processing 4 elements):
a) *O* (1) b) *O* (**n**/4) c) *O* (**n**/2) d) *O* (**n**) e) *O* (2**n**)

If the fence is at the tail of the list of length n, then to move to the node just before it, we have to move the `fence` to the `head` (one step) and then use `next()` n-1 times to move the fence to the desired node. Hence it takes 1 + (n-1)=n steps and therefore, answer c) is the correct one.

14. Finding an element in an array-based list with a given key value requires worst case time:
a) *O* (1) b) *O* (**n**/4) c) *O* (**n**/2) d) *O* (**n**) e) *O* (2**n**)

If the desired value is at the end of the array and to find an element we have to search starting at the top, then we would search n elements to find our value. Answer d) is the correct