資料結構與進階程式設計（**108-2**）

手寫作業八

B08705034 資管一 施芊羽

Question 1

Answer:

Array-based implementation:

```
template<class ItemType>
bool ArrayList<ItemType>::remove(int position)
{
    bool ableToRemove = (position >= 1) && (position <= itemCount);
    if (ableToRemove)
    { // delete item by shifting
        for (int fromIndex = position, toIndex = fromIndex-1;
        fromIndex < itemCount;
        fromIndex++, toIndex++)
            items[toIndex] = items[fromIndex];
            itemCount--; // decrease count of entries
    } // end if
    return ableToRemove;
} // end remove
```

In the `remove()` function of array-based list, what we do is keep moving the items of `ItemType` that previously stored in the array after the `position` which is going to be removed from `fromIndex` to `toIndex`. Therefore the time complexity of the `remove` function may be as $O(n)$, since at most $n-1$ items in the array need to be moved and it'll take nearly $3(n-1)$ steps to finishing the move and more time will required when `position` is smaller.

Link-based implementation:

```
template<class ItemType>
Bool LinkedList<ItemType>::remove(int position)
{
    bool ableToRemove = (position >=1) && (position <=itemCount)
    if( ableToRemove)
    {
        Node<ItemType>* curPtr = nullptr;
        if (position == 1)
        { // delete the first node from the list
            curPtr = headPtr;
            headPtr = headPtr->getNext();
        }
        else
        {
            Node<ItemType>* prevPtr = getNodeAt(postion - 1);
            curPtr = prevPtr->getNext();
            prevPtr->setNext(curPtr->getNext());
        }
        // return node to system
        curPtr->setNext(nullptr);
        delete curPtr;
        curPtr = nullptr;
        itemCount--;
    } // end if ableToRemove
    return ableToInsert;
}
```

In the `remove()` function of the link-based list, we'll first determine whether the `position` we want to remove is, if `position` is 1, it'll only require nine steps to finish removing; however, if `position` is not equal to `1`, we'll need to use the member function `getNodeAt` to find the pointer that point to the item at

Question 2

Answer:

(a) insert:

```
template<class ItemType>
bool ArrayList<ItemType>::insert(int newPosition, const ItemType& newEntry)
{
    bool ableToInsert = (newPosition >= 1) && (newPosition <= itemCount + 1) &&
(itemCount < maxItems);
    if (ableToInsert)
    { // make room for new item by shifting
        for (int pos = itemCount; pos >= newPosition; pos--)
            items[pos] = items[pos-1];
        // insert new item
        items[newPosition-1] = newEntry;
        itemCount++; // increase the size of the list by one
    } // end if
    return ableToInsert;
} // end insert
```

The time complexity of `insert()` might be $O(n)$. Since to insert a `newEntry` at `newPosition` requires to move all the items at and behind `newPosition` one unit behind, when the `newPosition` is 1, it means that all items that currently store in the `ArrayList` need to be moved. The maximum time will take $n + 4$ steps. While the `newPosition` is bigger, than the steps it takes will be less.

(b)remove:

```
template<class ItemType>
bool ArrayList<ItemType>::remove(int position)
{
    bool ableToRemove = (position >= 1) && (position <= itemCount);
    if (ableToRemove)
    { // delete item by shifting
        for (int fromIndex = position, toIndex = fromIndex-1;fromIndex < itemCou
nt; fromIndex++, toIndex++)
            items[toIndex] = items[fromIndex];
        itemCount--; // decrease count of entries
    } // end if
    return ableToRemove;
} // end remove
```

The time complexity of `remove()` will be $O(n)$. As mentioned in Question 1, the steps we do in `remove()` is move all the items that currently stored behind `position` one unit forward. Thus, when `position` is 1, then all the other $n - 1$ items need to be move. It'll take $n + 2$ steps to finish it. When the `position` is bigger, the steps it requires will decrease.

(c)retrieve:

```
template<class ItemType>
ItemType ArrayList<ItemType>::getEntry(int position) const throw(PrecondViolated
Excep)
{
    bool ableToGet = (position >=1) && (position <= itemCount);
    if (ableToGet)
        return items[position – 1];
    else
    {
        string message = "getEntry() called with an empty list or ";
        message = message + "invalid position.";
        throw(PrecondViolatedExcep(message));
    }
}
```

The time complexity to retrieve an entry is equal to the time complexity of the member function `getEntry()` which is $O(1)$. Since if `position` is able to get, it'll only require 3 steps to retreive the item, if not it'll take 3 more steps to throw the error message.

(d)find:

```
template<class ItemType>
int ArrayList<ItemType>::getIndexOf(const ItemType& Entry) const throw(logic_err
or)
{
    bool found = false;
    for(int i = 1 ; i <= itemCount ; i++){
        if(items[i] == Entry){
            found = true;
            return i;
        }
    }
    if(found == false){
        throw(logic_error("Cannot find it!"))
    }
}
```

The time complexity to find an item is $O(n)$. To find where an entry is all we need to do is to run a `for` loop to search for the entry and return its index. The more time will be required if the entry is stored in more back side of the list. If the item is stored as the last item of the list, it'll take $3n+4$ steps to finish the `find` function; if it doesn't in the list, it'll take $3n+4$ steps. If the item stores in the front, it'll save more time.

Question 3

Answer:

(a)insert:

```
template<class ItemType>
Node<ItemType>* LinkedList<ItemType>::getNodeAt(int position) const
{
    assert ( (position >= 1) && (position <= itemCount) );
    ListNode* curPtr = headPtr;
    for (int skip = 1; skip < position; skip++)
        curPtr = curPtr->getNext();
    return curPtr;
}
template<class ItemType>
bool LinkedList<ItemType>::insert(int newPosition, const ItemType& newEntry)
{
    bool ableToInsert = (newPosition >= 1) && (newPosition <= itemCount+1);
    if ( ableToInsert )
    { // create a new node
        Node<ItemType>* newNodePtr = new Node<ItemType>(newEntry);
        // attach new node to chain
        if (newPosition == 1)
        { // insert new node at beginning of chain
            newNodePtr->setNext(headPtr);
            headPtr = newNodePtr;
        }
        else
        {
            Node<ItemType> *prevPtr = getNodeAt(newPosition-1);
            newNodePtr->setNext(prevPtr->getNext());
            prevPtr->setNext(newNodePtr);
        }
        itemCount++;
    } // end if ableToInsert
    return ableToInsert;
}
```

The time complexity of `insert()` is $O(n)$. If the `newPosition` is equal to 1, it only requires six steps to finish the `insert()`; however, if it's not 1, it'll require $O(n)$ to finish the `insert()` because the function needs to get access of the `getNodeAt()` function to do the rest of the insertion, while the rest of the functions is $O(1)$, the `getNodeAt()` is $O(n)$, because it'll take at most $n+2$ steps(which occur when `position=itemCount`) to find its node. Therefore, the time complexity of `insert()` is $O(n)$.

(b)remove:

```
template<class ItemType>
Bool LinkedList<ItemType>::remove(int position)
{
    bool ableToRemove = (position >=1) && (position <=itemCount)
    if( ableToRemove)
    {
        Node<ItemType>* curPtr = nullptr;
        if (position == 1)
        { // delete the first node from the list
            curPtr = headPtr;
            headPtr = headPtr->getNext();
        }
        else
        {
            Node<ItemType>* prevPtr = getNodeAt(postion - 1);
            curPtr = prevPtr->getNext();
            prevPtr->setNext(curPtr->getNext());
        }
        // return node to system
        curPtr->setNext(nullptr);
        delete curPtr;
        curPtr = nullptr;
        itemCount--;
    } // end if ableToRemove
    return ableToInsert;
}
```

The time complexity of `remove()` is $O(n)$. Just like the `insert()` function, when `position` is equal to 1, the time complexity will be $O(n)$ since it takes at most ten steps to finish the function; however, if not, it'll need to use `getNodeAt()` function again and it'll make the whole function has a time complexity of `O(n)` .

(c)retrieve:

```
template<class ItemType>
ItemType LinkedList<ItemType>:getEntry(int position) const throw(PrecondViolated
Excep)
{
    bool ableToGet = (position >= 1) && (position <= itemCount);
    if(ableToGet)
    {
        Node<ItemType>* nodePtr = getNodeAt(position);
        return nodePtr->getItem();
    }
    else {
        string message = “getEntry() called with an empty list or”
        message = message + “invalid position.”;
        throw(PrecondViolatedExcep(message));
    }
}
```

The time complexity to retrieve an item from the list will be the same as the function `getEntry()` take which will be $O(n)$. In the `getEntry()` function, if `ableToGet` is `true` , it'll require to use `getNodeAt()` function which has a time complexity of $O(n)$ and the rest of the function is of $O(1)$, that'll result the whole function has a $O(n)$ time complexity; if `false` , the whole function only requires five steps. Therefore, the time complexity to retrieve is $O(n)$.

(d)find:

```
template<class ItemType>
int LinkedList<ItemType>::getIndexOf(const ItemType& Entry) const throw(logic_er
ror)
{
    bool found = false;
    int i = 1;
    Node<ItemType> curPtr = headptr;
    while(curPtr != nullptr){
        if(curPtr->getItem() == Entry){
            found = true;
            return i;
        }
        i++;
        curPtr = curPtr->getNext();
    }
    if(found == false){
        throw(logic_error("Cannot find it!"))
    }
}
```

The time complexity to find the index of an entry will be $O(n)$ just as the above function `getIndexOf()` takes. Just like to find an entry in an array-based array, the function need to look through all the items that store in the list which will at most $O(n)$. The maximum steps required in the above function will be $3n + 5$ which occurs when the given `Entry` cannot be found in the list. Hence the time complexity of the function can be noted as $O(n)$.

Question 4

Answer:

```
LinkedSortedList<Itemtype>* MergeSort(const LinkSortedList<Itemtype>& ListA , co
nst LinkSortedList<Itemtype>& ListB):
        LinkedSortedList<Itemtype>* mergeList;
        create a currentAPtr which point to the first item of ListA;
        create a currentBPtr which point to the first item of ListB;
        while currentAPtr and currentBPtr are not nullptr:
            if currentAPtr->getItem() < currentBPtr->getItem() or currentBPtr =
nullptr:
                mergeList.insertSorted(currentAPtr->getItem())
                currentAPtr = currentAPtr->getNext()
            else:
                mergeList.insertSorted(currentBPtr->getItem())
                currentBPtr = currentBPtr->getNext()
        return mergeList;
```

Solution: I make the two sorted list named `ListA` and `ListB` to be the parameters of the `MergeSort()` function. In the function, I'll keep compare both items which are not yet been put into the new list that in the front of both `ListA` and `ListB` , and use the function `insertSorted()` to add the item which will save time. After all the items have been put into the new third list, I'll return the pointer point to new third list.