Camille Scott
CSSAP 443
MP1 – Design Documentation

a)

TmemoryManager's Constructor

	Worst and Average Case	
	Cost	**Times**
`FreeLinksList <- get TmyList object reference (call to TmyList())`	1	1
`pointer index <- 0`	1	1
`while( index LESS THAN MaxListSize-SizeOfLink )`	1	MaxListSize/SizeOfLink (see below)
`      call TmyList's  insertElement(–1, index) for FreeLinkList's`	1	MaxListSize/SizeOfLink
`      index <- index + SizeOfLink`	1	MaxListSize/SizeOfLink

Running Time Calculation

(MaxListSize-SizeOfLink)/SizeOfLink

= MaxListSize/SizeOfLink

index = 0

TmyList's Constructor

	Worst and Average Case	
	Cost	**Times**
`head <- tail <- null`	1	1

TmyList's public void insertElement(int key, int freeLinkAddress)

```
// If this is really a free link, set the key.  If this is an empty
// list set the head to point to the new link, else add the item to
// the end of the existing list.  After adding the item set the tail
// pointer and set the new item's next pointer to null
```

	Worst Case	
	Cost	**Time**
`if( freeLinkAddress NOT EQUAL null )`	1	1
`      dereference freeLinkAddress <- key`	1	1
`      if( listIsEmpty())`	1	1
`            head <- freeLinkAddress`	1	1/2
`      else`		
`            dereference tail's next <- freeLinkAddress`	1	1/2
`      tail <- freeLinkAddress`	1	1
`      dereference tail's next <- null`	1	1

To determine the worst case running time of TmemoryManager(), we must take into account the running time of TmyList() and insertElement(…).

TmyList()'s T(n) = 1*1 =1

insertElement ()'s T(n) = (1*1) + (1*1) + (1*1) + (1*1/2) + (1*1/2) + (1*1) + (1*1) = 1+1+1+1/2+1/2+1+ 1 = 6

TmemoryManager's T(n) = (TmyList()'s T(n) * (1*1)) + (1*1) + (1* MaxListSize/SizeOfLink) +
(insertElement ()'s T(n)*(1* MaxListSize/SizeOfLink)) + 1*MaxListSize/SizeOfLink)

TmemoryManager's T(n) = (1 * 1) + 1 + MaxListSize/SizeOfLink + (6*MaxListSize/SizeOfLink) + MaxListSize/SizeOfLink

TmemoryManager’s T(n) = 2*MaxListSize/SizeOfLink + 2
O(MaxListSize/SizeOfLink)

The average and worst case run time is the same, because you are guaranteed to execute the statements based on MaxListSize and the SizeOfLink.

In our case MaxListSize = 30

index :	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
value :	-1	2	–1	4	–1	6	–1	8	–1	10	–1	12	–1	14	–1	16	–1	18	–1	20	–1	22	–1	24	–1	26	–1	28	–1	-1
key/next:	k	n	k	n	k	n	k	n	k	n	k	n	k	n	k	n	k	n	k	n	k	n	k	n	k	n	k	n	k	n

b)

TmyList’s int removeLinkFromList(int key)

```
pointer previous <- null
pointer current  <- head

// if match is first element, reset fHead, remove and return link
if(( NOT listIsEmpty()) AND ( key EQUALS dereference of current ))
      call removeFirstLinkFromList()
      return current

// not found so, walk through the rest of the list
previous <- current
current  <- dereference current’s next
while( current NOT EQUAL null )
      // If a match is found, set the previous pointer's next pointer
      // to the value of current's next pointer, therefore unlinking
      // current.  If current was the last link then reset the tail
      // pointer and return the unlinked value
      if( key EQUALS dereference of current)
            dereference previous’ next <- dereference current’s next
            if( dereference previous’ next EQUALS null )
                  tail <- null
                  return current
            previous <- current
            current  <- dereference current’s next

// if key is not found, an invalid address is returned.
// and the content of the list is not altered
return current
```

TmyList’s public void insertElement(int key, int freeLinkAddress)

```
// If this is really a free link, set the key.  If this is an empty
// list set the head to point to the new link, else add the item to
```

```
// the end of the existing list.  After adding the item set the tail
// pointer and set the new item's next pointer to null
if( freeLinkAddress NOT EQUAL null )
      dereference freeLinkAddress <- key
      if( listIsEmpty())
            head <- freeLinkAddress
      else
            dereference tail’s next <- freeLinkAddress
      tail <- freeLinkAddress
      dereference tail’s next <- null
```

c)

TmemoryManager’s public int nextFreeLink()

Code	Worst Case Cost	Worst Case Time
`// returns the address of the next free linkEntry`		
`// if there are no more free ones, return kInvalidAddress.`		
`return result of calling TmyList’s removeFirstLinkFromList() for FreeLinksList`	1	1

TmyList’s public int removeFirstLinkFromList()

Code	Worst Case Cost	Worst Case Time
`// delete the first link from the list, if list is empty`		
`// an invalid address is returned.`		
`pointer current <- head`	1	1
`if( NOT listIsEmpty())`	1	1
`      head <- dereference current’s next`	1	1
`      // if first and only element then reset tail`		
`      if( listIsEmpty())`	1	1
`            tail <- null`	1	1
`return current`	1	1

Running Time Calculation

To determine the worst case running time of nextFreeLink(), we must take into account the running time of removeFirstLinkFromList().

removeFirstLinkFromList()’s T(n) = (1*1) + (1*1) + (1*1) + (1*1) + (1*1) + (1*1) = 1+1+1+1+1+1=6

nextFreeLink()’s T(n) = removeFirstLinkFromList()’s T(n) * (1*1) + = 1 * 6 = 6

O(1)

TmemoryManager’s public void returnFreeLink(int linkAddress)

Code	Worst Case Cost	Worst Case Time
`// linkAddress is a linkEntry that is not being used anymore.`		
`// add the free element to the free Linked List`		
`call TmyList’s insertElement(–1, linkAddress) for FreeLinkList`	1	1

TmyList’s public void insertElement(int key, int freeLinkAddress)
See part a) above for analysis of insertElement(…)

Running Time Calculation

To determine the worst case running time of returnFreeLink(), we must take into account the running time of insertElement().
insertElement ()'s T(n) = (1*1) + (1*1) + (1*1) + (1*1/2) + (1*1/2) + (1*1) + (1*1) = 1+1+1+1/2+1/2+1+ 1 = 6
returnFreeLink()'s T(n) = insertElement ()'s T(n) * (1*1) + = 6 * 1 = 6
O(1)

d)

TmyQueue's Constructor	Worst Case Cost	Worst Case Time
`fQueue <- get TmyList object reference (call to TmyList())`	1	1

See part a) for TmyList() Analysis

T(n) = TmyList()'s T(n) * (1*1) = 1* 1 = 1
O(1)

TmyQueue's public void enQueue(int key, int freeLinkAddress)	Worst Case Cost	Worst Case Time
`call insertElement(key, freeLinkAddress) for the queue`	1	1

See part a) above for analysis of insertElement(…)

Running Time Calculation

To determine the worst case running time of enQueue(), we must take into account the running time of insertElement().
insertElement()'s T(n) = (1*1) + (1*1) + (1*1) + (1*1/2) + (1*1/2) + (1*1) + (1*1) = 1+1+1+1/2+1/2+1+1 = 6
T(n) = insertElement()'s T(n) *(1*1) = 6 * 1 = 6
O(1)

TmyQueue's public int deQueue()	Worst Case Cost	Worst Case Time
`return the result of calling TmyList's removeFirstLinkFromList()`	1	1

See part c) for analysis of removeFirstLinkFromList()
To determine the worst case running time of deQueue(), we must take into account the running time of removeFirstLinkFromList ().
removeFirstLinkFromList()'s T(n) = (1*1) + (1*1) + (1*1) + (1*1) + (1*1) + (1*1) = 1+1+1+1+1+1=6
T(n) = removeFirstLinkFromList()'s T(n) * (1*1) = 6 * 1 = 6
O(1)