Linked Lists Data Structure

Chapter 12

Pages 680-682

Mostly These Notes

Motivation

- Arrays have a major performance limitation
  - Adding an additional element is time consuming

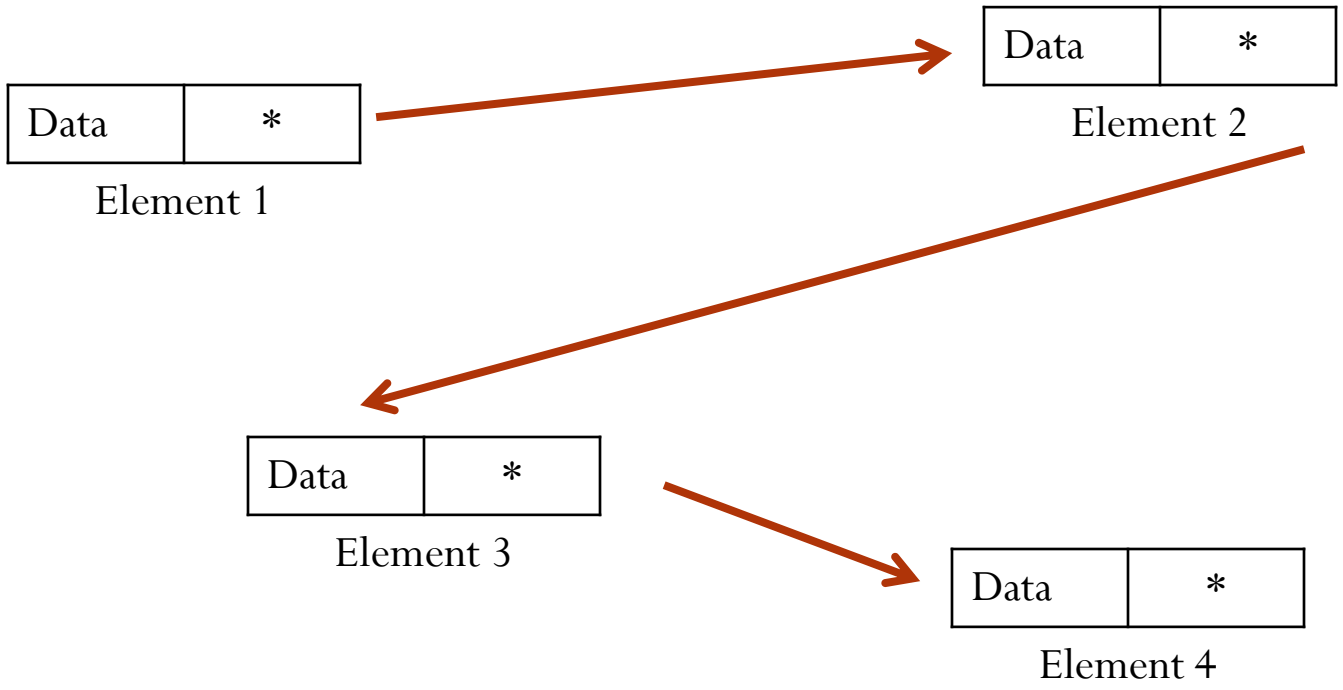
- Review of Adding an Element
  - Allocate enough space for our new array
  - Copy the contents

- The time it takes to resize the array is proportional to the size of the array
  - Therefore: Adding an element is $\Theta(n)$ where *n* is the number of elements

Motivation

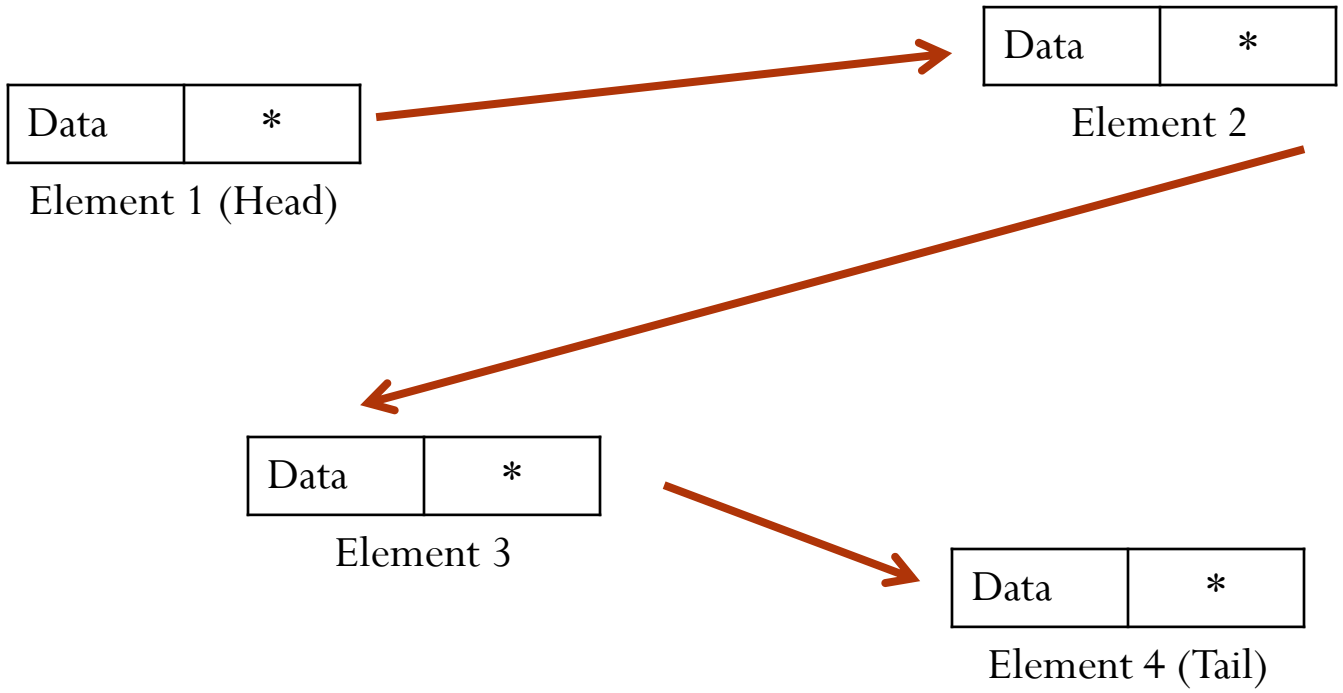
- Arrays perform poorly when the size of them is constantly being adjusted
  - Grocery List
  - Team Roster
  - Dynamic Queues
  - Etc.
- They are also terrible for unbalanced trees which is another story
- Can we do better?

Linked Lists



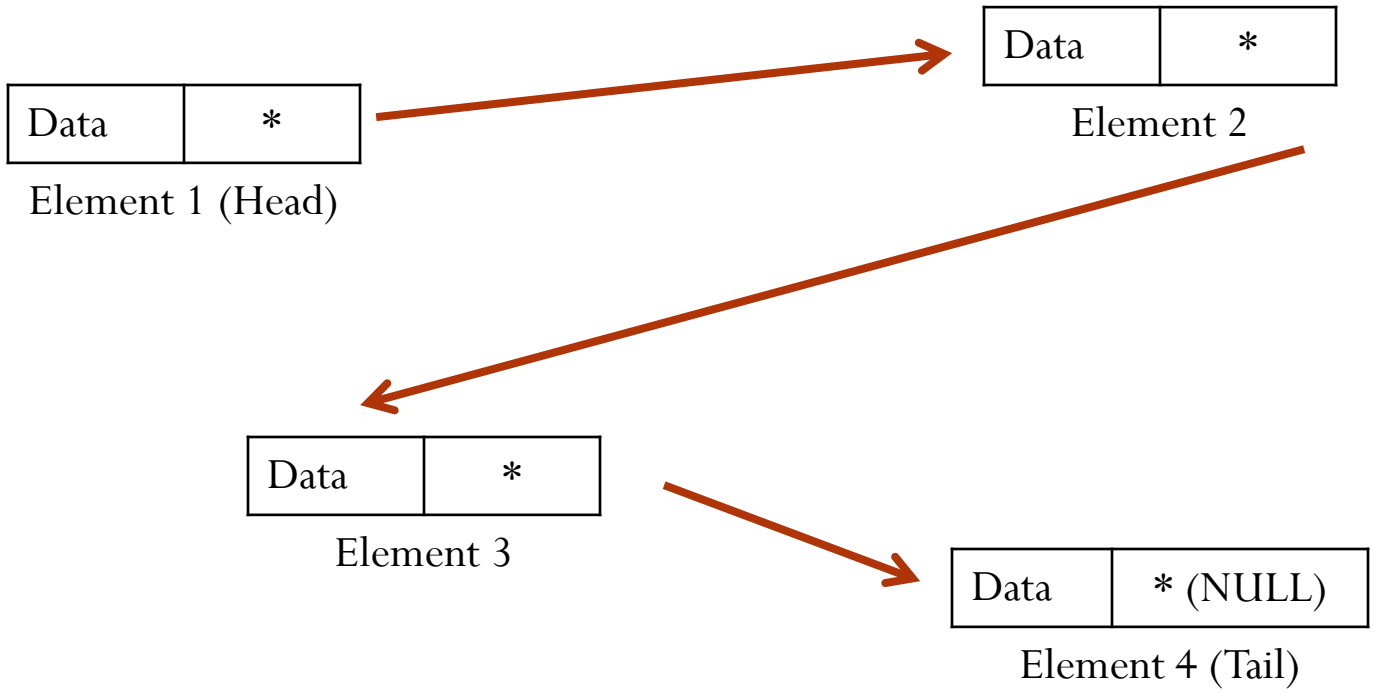
- Each element in the list has a link to the next element in the list

Linked Lists



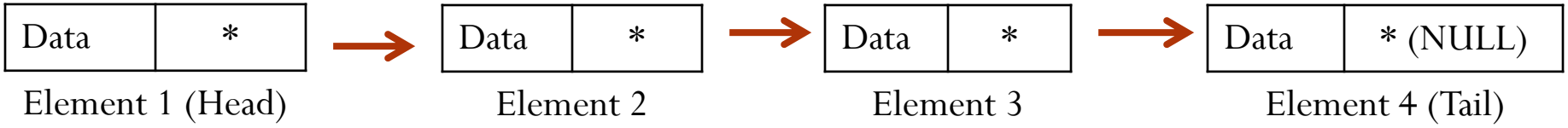
- The first element is the Head
- The last element is the Tail

Linked Lists



- The tail's pointer points to NULL
- NULL is an exist that doesn't exist (equivalent to 0)

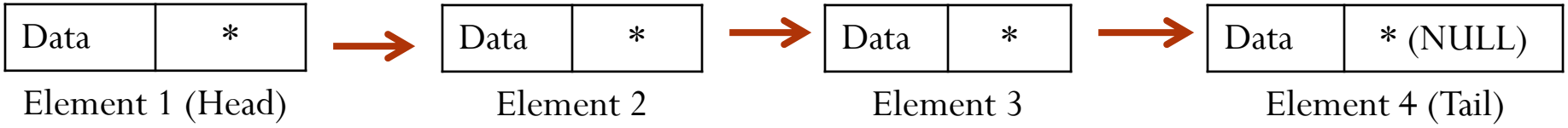
Linked Lists (Traversal)



- Notice each element only knows the location of the next element
  - From a given element I can always access the next element but not the previous
- A full traversal across the list can be achieved by starting at the *Head* and hopping across each element to the *Tail*
- Therefore from the Head it is possible to access all elements (eventually)

- As such only the Head of the linked list is stored on the stack

Linked Lists (Traversal)

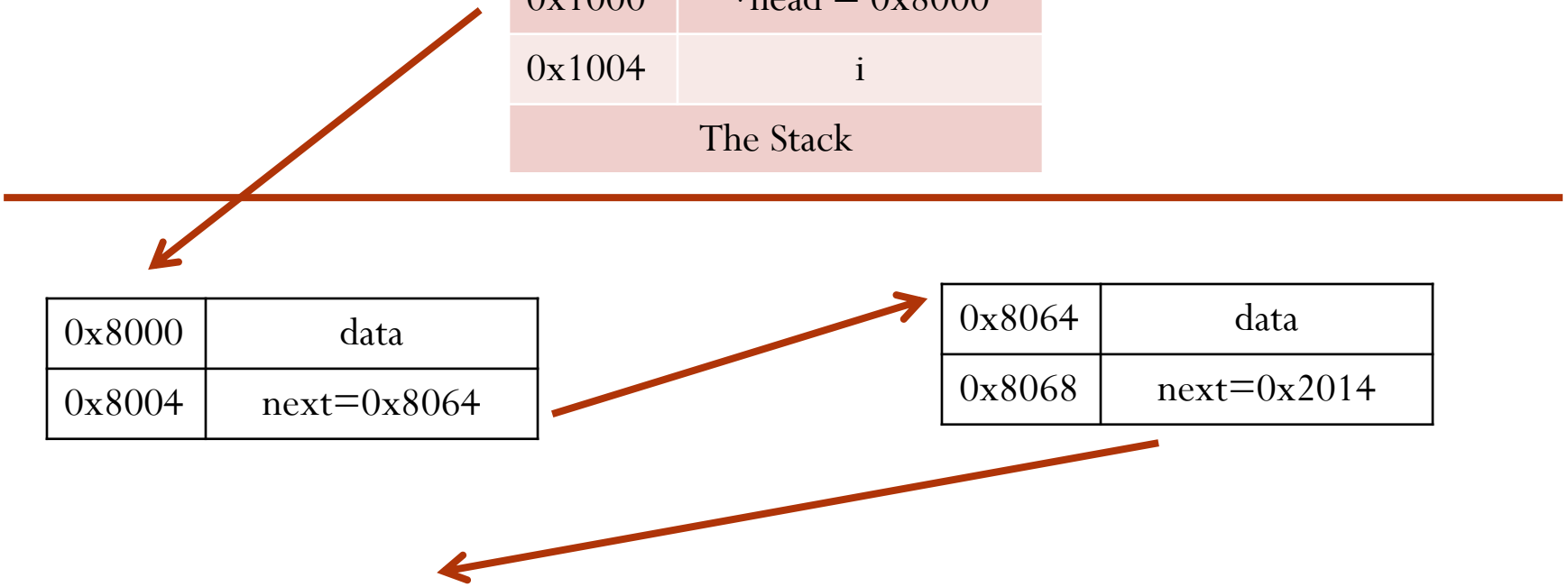


- If I only keep the head do I have direct access to each element in the list?
  - No.

- Only indirect access:
  - To access the tail element I must travel all previous nodes

Linked Lists in Memory

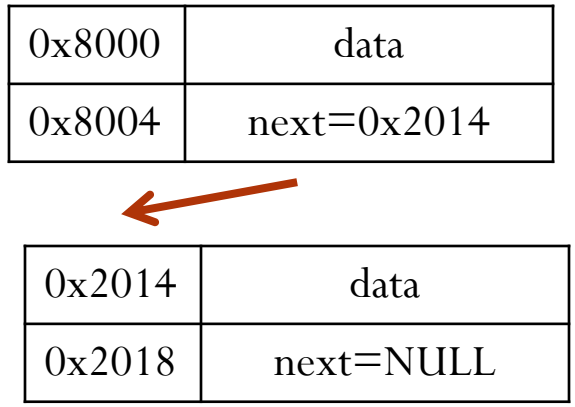
Mem Addr	Variable
0x1000	*head = 0x8000
0x1004	i
The Stack	



Adding An Element

Mem Addr	Variable
0x1000	*head = 0x8000
0x1004	*newElement = NULL
The Stack	

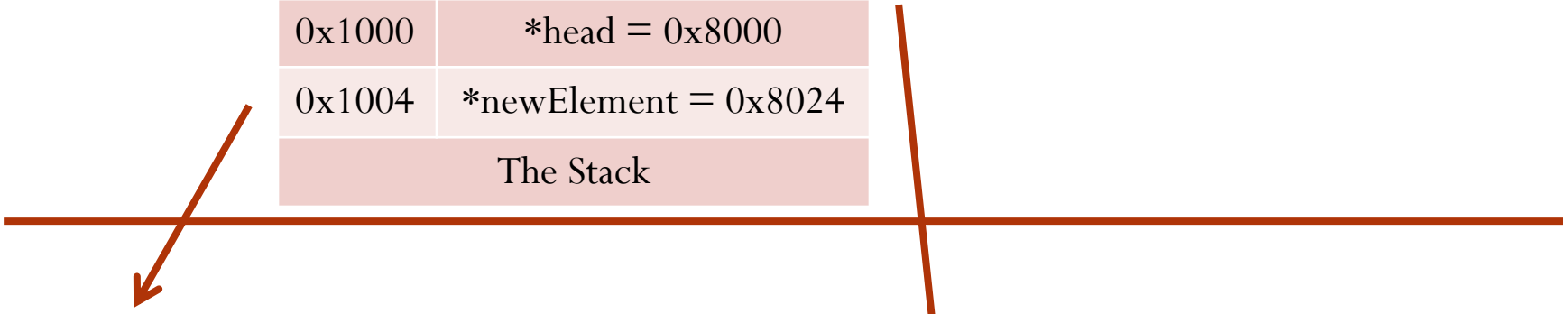
- Starting list
- Two elements
  - Head and Tail



Adding An Element

- Allocate a new node in the list

Mem Addr	Variable
0x1000	*head = 0x8000
0x1004	*newElement = 0x8024
The Stack	



0x8024	data
0x8028	next=

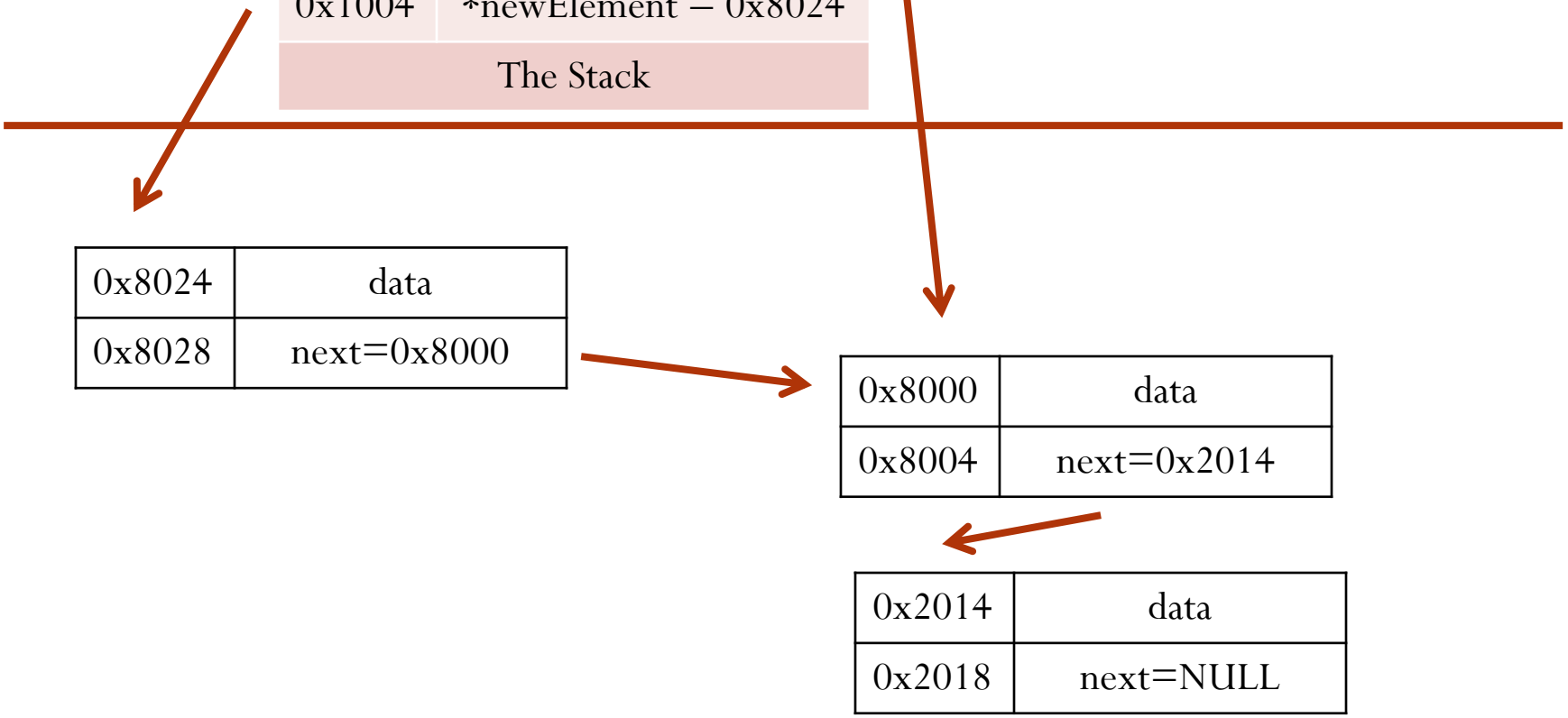
0x8000	data
0x8004	next=0x2014

0x2014	data
0x2018	next=NULL

Adding An Element

- Set the pointer of new element to the Head of the list

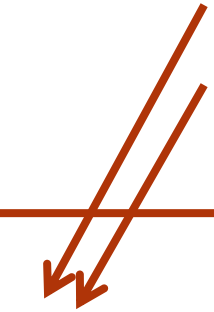
Mem Addr	Variable
0x1000	*head = 0x8000
0x1004	*newElement = 0x8024
The Stack	



Adding An Element

- Set the head pointer to point to the new element

Mem Addr	Variable
0x1000	*head = 0x8024
0x1004	*newElement = 0x8024
The Stack	



0x8024	data
0x8028	next=0x8000

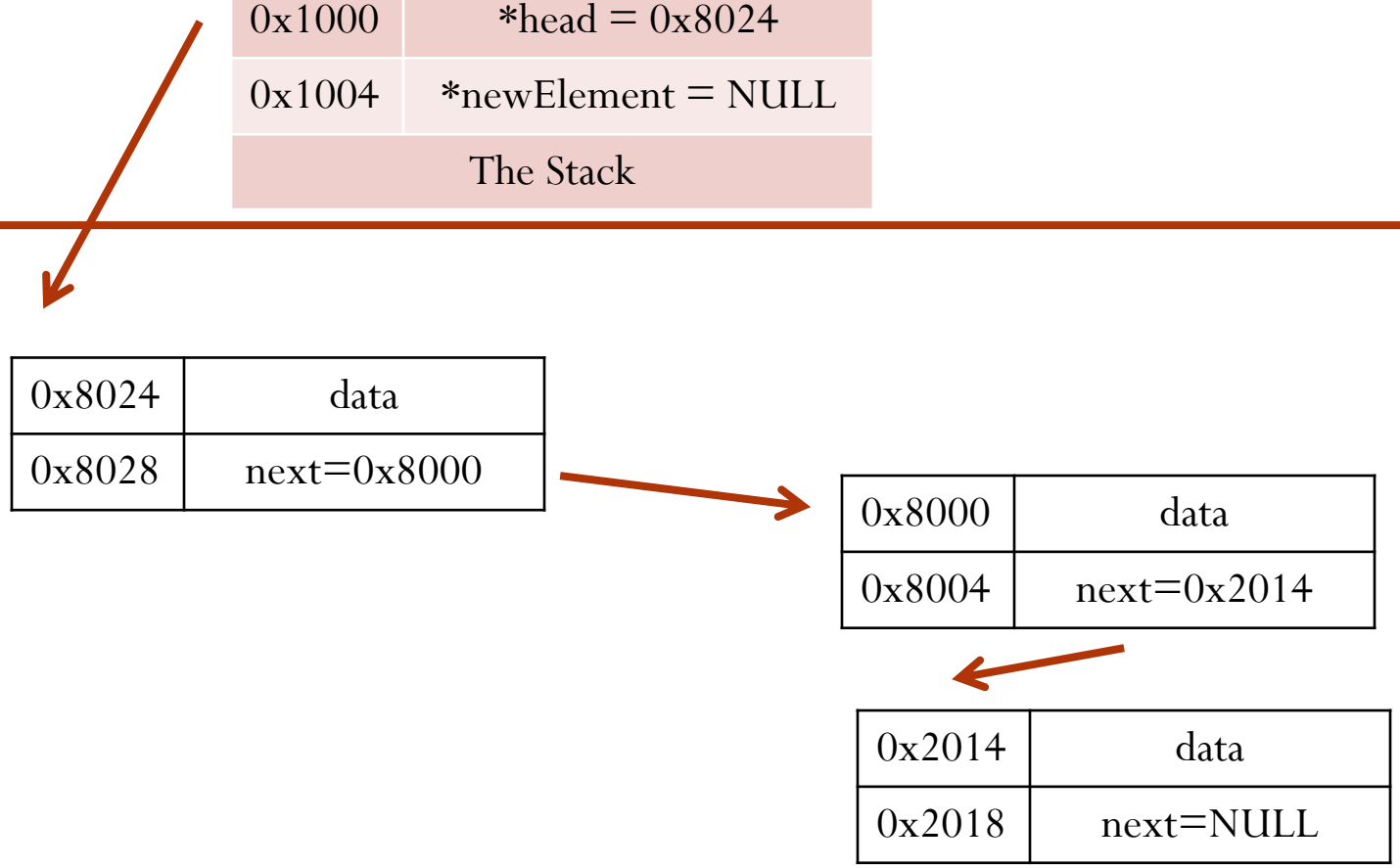
0x8000	data
0x8004	next=0x2014

0x2014	data
0x2018	next=NULL

Adding An Element

- Now we are free to use newElement again and repeat the process

Mem Addr	Variable
0x1000	*head = 0x8024
0x1004	*newElement = NULL
The Stack	



Adding to Linked List

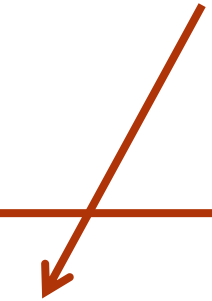
- Allocation of a new element is done one at a time
- New elements can be inserted at the start of the list
  - This operation is independent of the size of the linked list
  - Constant time operation

- Elements can also be inserted at any point in the list but the time of the operation will be proportional to the number of elements traversed

Deleting An Element

- Lets delete the first element in the list

Mem Addr	Variable
0x1000	*head = 0x8024
0x1004	*tmp= NULL
The Stack	



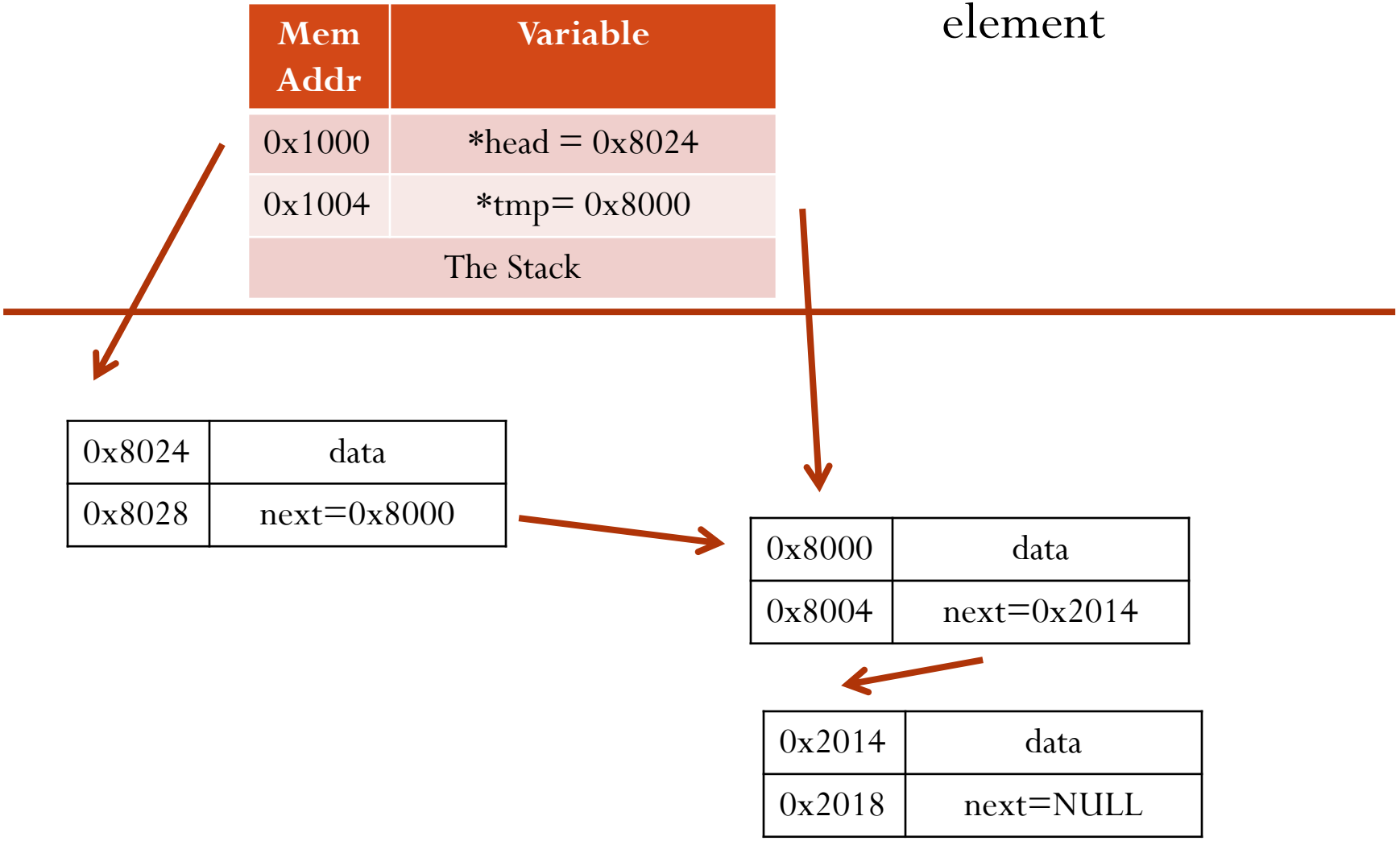
0x8024	data
0x8028	next=0x8000

0x8000	data
0x8004	next=0x2014

0x2014	data
0x2018	next=NULL

Deleting An Element

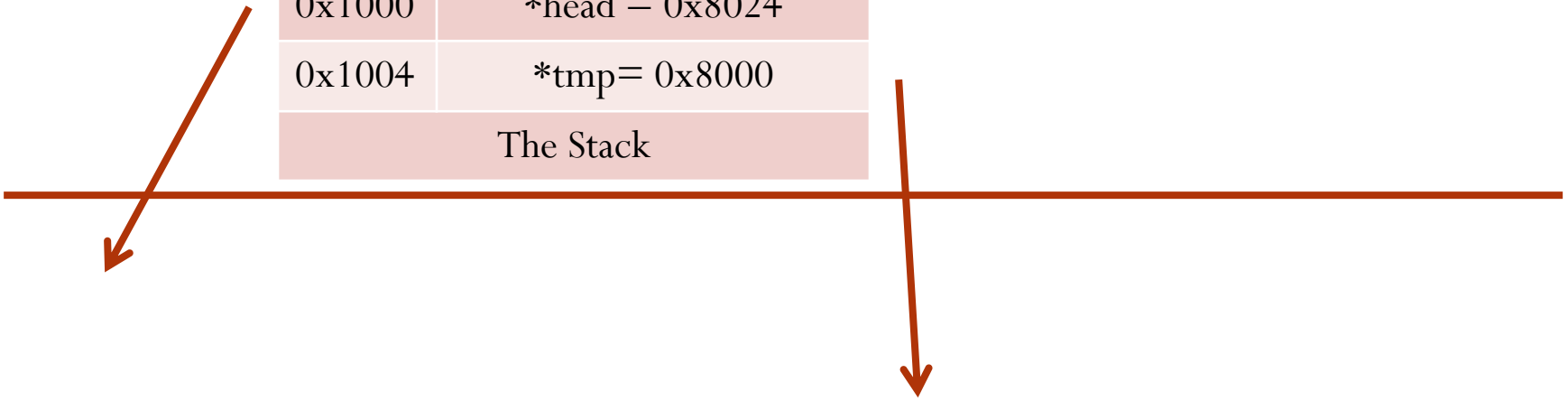
- Store the next pointer of the head element



Deleting An Element

- Deallocate the head element

Mem Addr	Variable
0x1000	*head = 0x8024
0x1004	*tmp= 0x8000
The Stack	



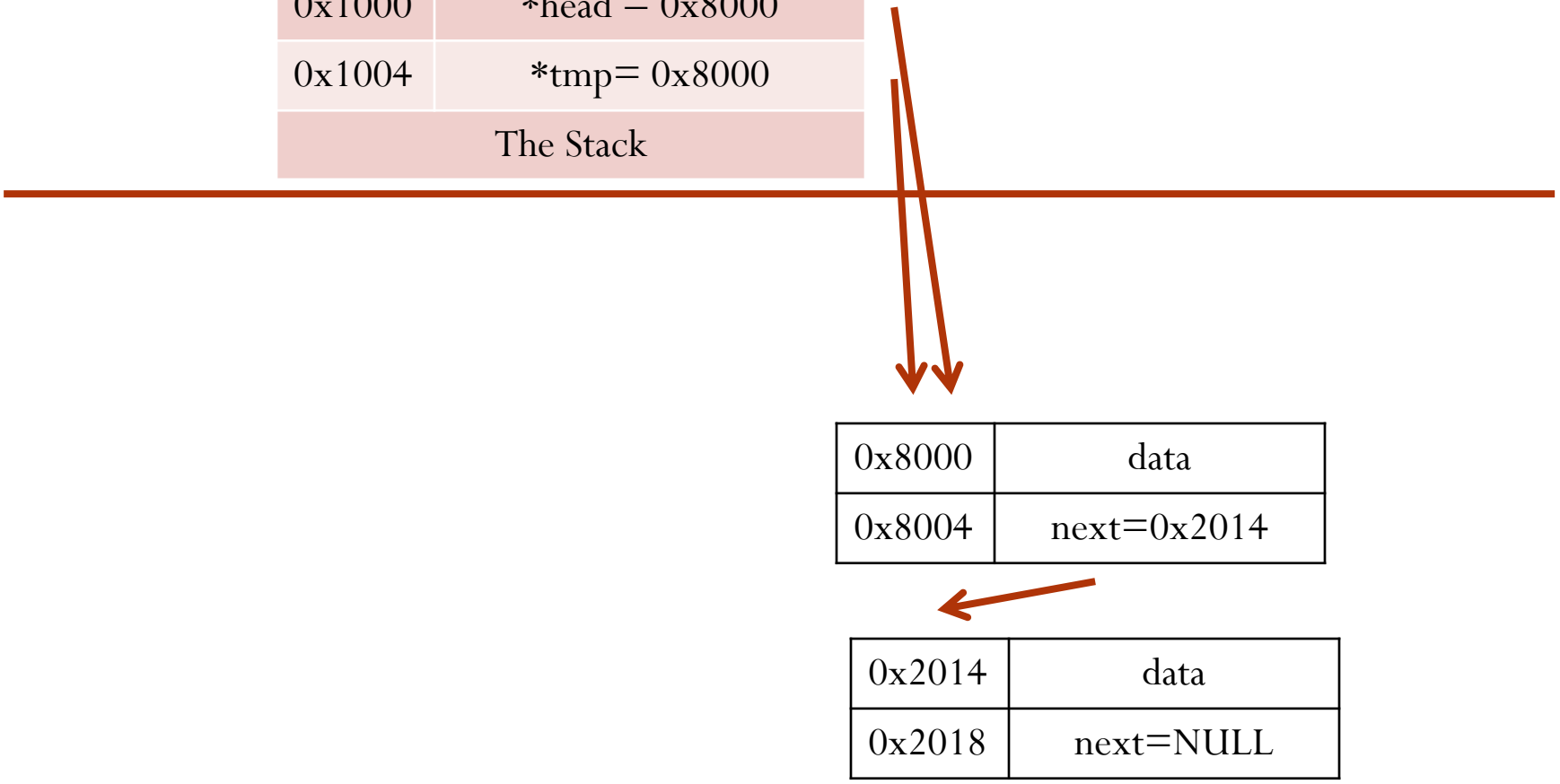
0x8000	data
0x8004	next=0x2014

0x2014	data
0x2018	next=NULL

Deleting An Element

- Set the head to point to the same element as the tmp

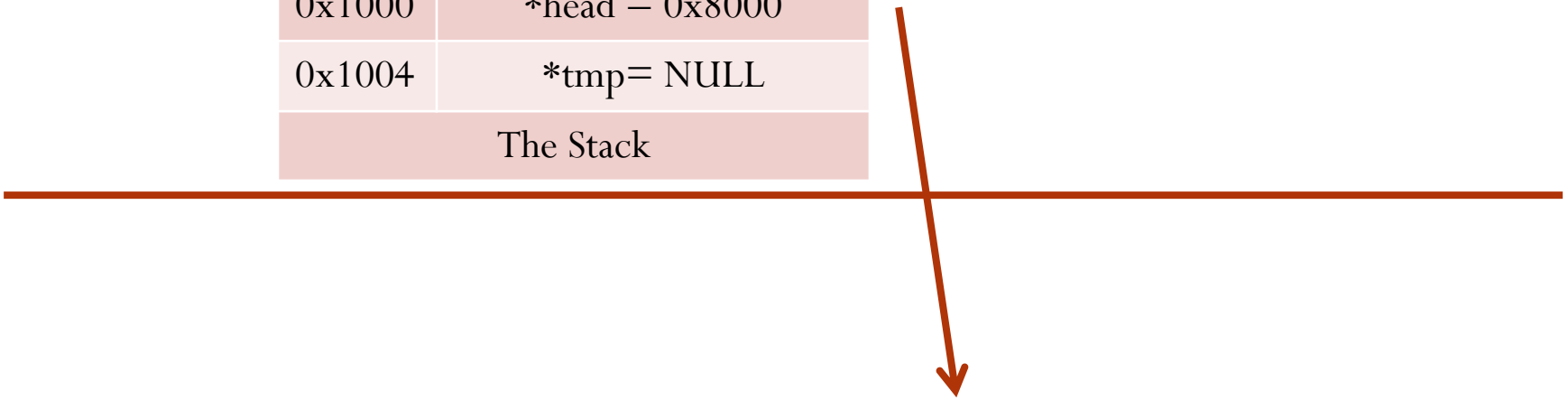
Mem Addr	Variable
0x1000	*head = 0x8000
0x1004	*tmp= 0x8000
The Stack	



Deleting An Element

- Tmp can now be reused and the process can be repeated

Mem Addr	Variable
0x1000	*head = 0x8000
0x1004	*tmp= NULL
The Stack	



0x8000	data
0x8004	next=0x2014

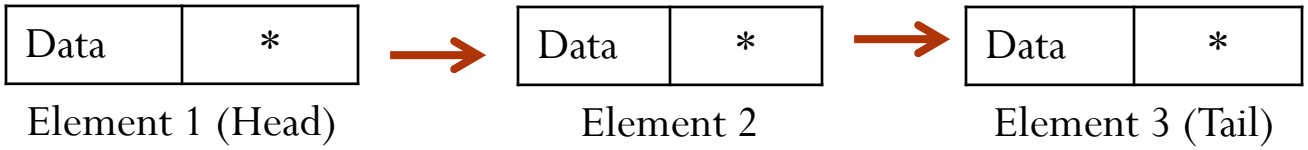
0x2014	data
0x2018	next=NULL

Deleting an Element from a List

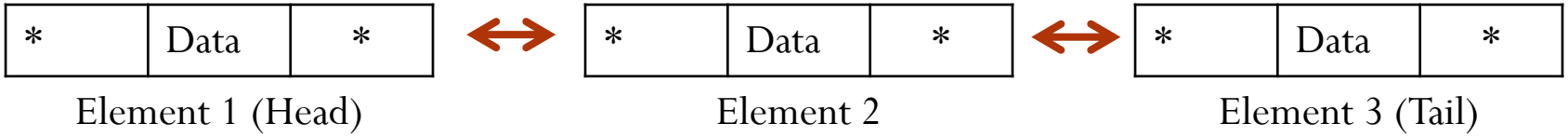
- In general, any element in the list can be deleted regardless of its location
- The time for deletion is proportional to the number of elements traversed to reach the element to be deleted
- Thus deleting at the start of the list is very efficent

Linked List Styles

Singly Linked Lists

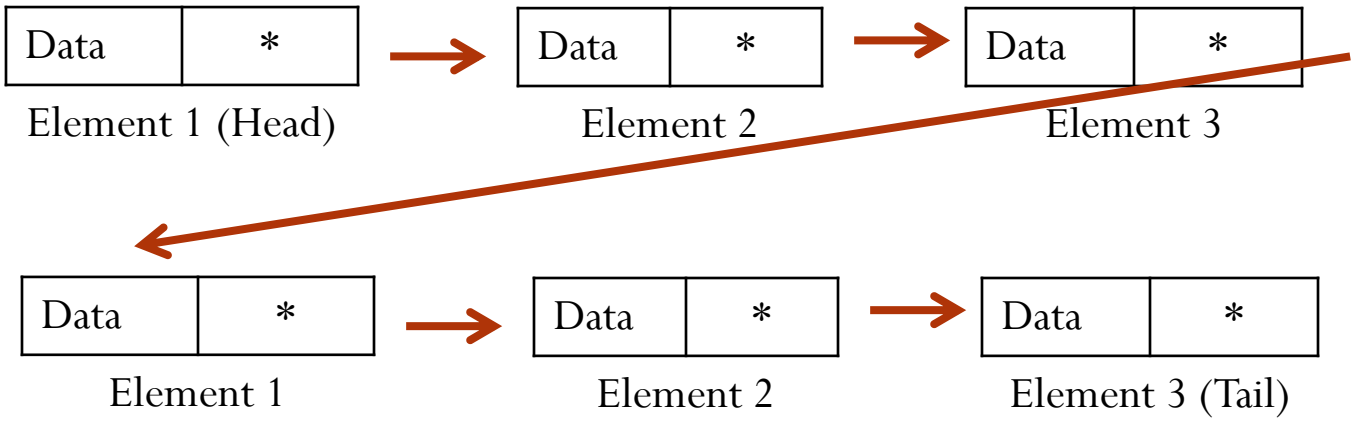


Doubly Linked Lists



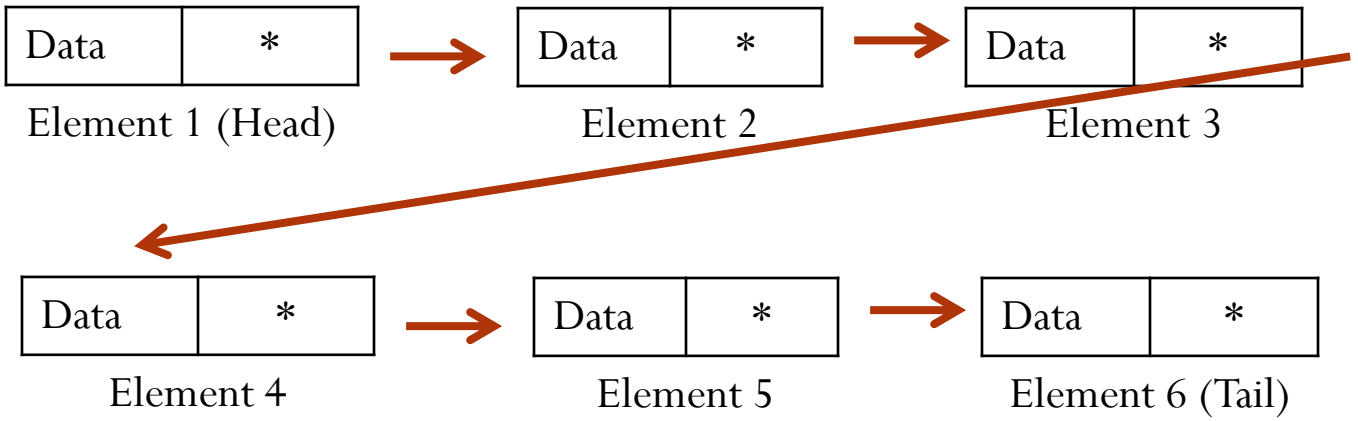
- Doubly linked lists allow for traversal both backwards and forwards

Linked Lists Joining Linked Lists



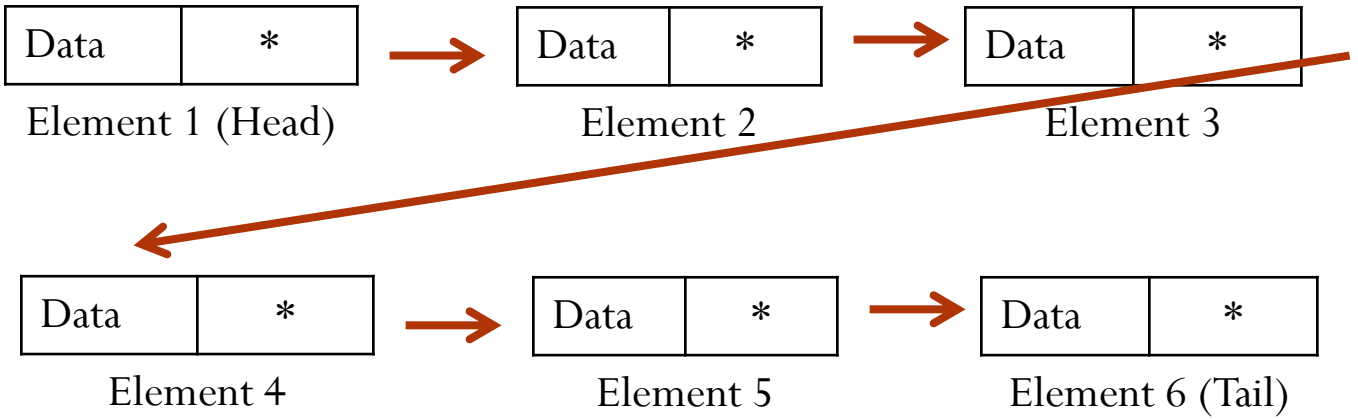
- Point the tail of one list to the head of the other

Linked Lists Joining Linked Lists



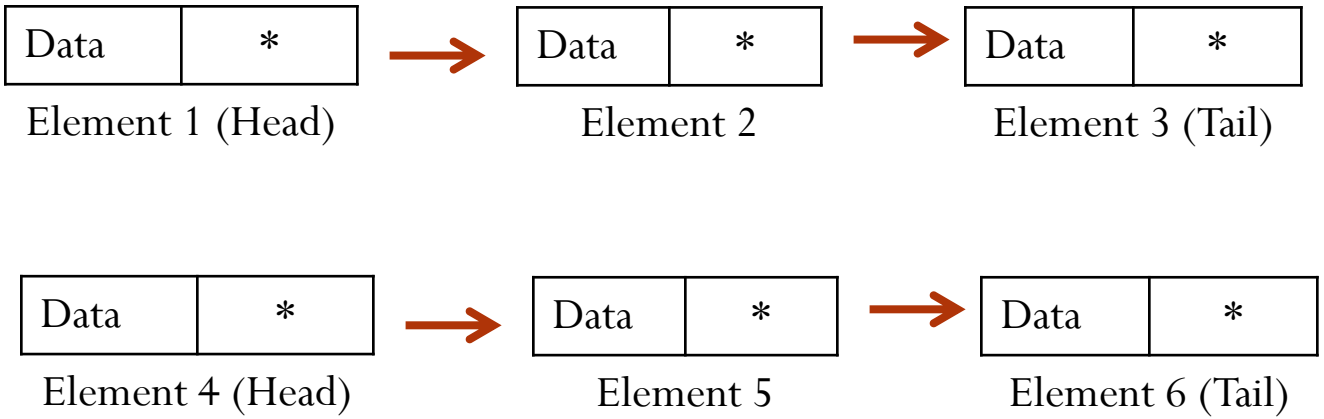
- If the tail is tracked as well as the head it is a constant time operation
- Otherwise the time taken is equal to the size of one of the lists
- In practice tail is tracked too
  - Many operations are sped up by tracking the tail
  - Memory is cheap

Splitting Linked List



- Make a new list starting at Element 4

Splitting Linked List



- Save the a pointer to element 4 as the new head
- Remove the pointer of the previous element (i-1) to element i
- Time is proportional to the number of elements traversed

Memory Efficiency

- For each node of the list we must have at least one pointer to another node
- Lets take a linked list of integers
  - The memory footprint would be twice that of an equivalent array!!!!
- If each node contains a complex structure then the memory footprint will be less than twice that of an equivalent array but is still guaranteed to be larger
- Thus linked lists have a memory overhead

Memory Efficiency

- In practice Linked Lists are often more memory efficient
- Dynamic arrays are rarely sized to the exact # of elements
  - This is done to mitigate the number of reallocations
- Typically nodes are made of complex structures where the memory taken up by the pointer is insignificant

Performance Considerations

- Is traversing the list any slower than traversing an array from start element to end element?
  - Absolutely

- Iterating an array is a simple arithmetic operation
- Iterating a linked list requires a dereference operation and a pointer update

- Arrays are sequential in memory and therefore have good cache performance
- Linked lists are not sequential and are bound to generate many cache misses costing clock cycles

Data Structure Comparison

Action	Array	Linked List
Add Item	Large allocation and copy elements $\Theta(n)$	Assuming front, Constant time $\Theta(c)$
Random Access:	Direct Access $\Theta(c)$	All previous elements must be traversed $\Theta(n)$
Joining:	Large allocation and copy all elements from both arrays $\Theta(n)$	Join tail to head, assuming tail is tracked $\Theta(c)$
Splitting:	Two allocations and copy all elements from both arrays $\Theta(n)$	All elements up till split point must be traversed. $\Theta(n)$. Though same time complexity it will be faster
Iterating	Faster operation because of sequential memory	Cache misses, more complex of an operation

- $\Theta(c)$: Constant time, independent of elements
- $\Theta(n)$: Linear Time to the number of elements
- Constant time is faster than linear time

Structure of Linked List

```
struct Node {
   string itemName;
   Node *next;
};
```

- Each node in the list must be structure
- Each node must have a pointer to another node
- The structure can be as large as needed

Adding a Node

```
Node* addItem(Node *list, string itemName)
{
    Node *newItem = new Node; // gen new node
    newItem->itemName = itemName;
    newItem->next = list; // point it to old head
    return newItem; // return new head
}
```

1. Allocate new node
2. Set data
3. Set to point to original head
4. Return pointer of new head

Printing the List

```
void printList(Node *list)
{
    while(list != NULL)
    {
       cout << list->itemName << endl;
       list = list->next;
    }
}
```

- Traversal of the nodes
- Start at the head
- Set current pointer to the next pointer
- Stop when no more nodes (NULL)

Removing Head Node

```
Node* removeHead(Node *list)
{
    // Save pointer of next element
    Node *head = list->next;
    delete list; // delete head element
    return head; // return new head
}
```

- Store the pointer of the node after the head
- Deallocate head node
- Return new head node

Example of Using List

```
Node *myList = NULL;
myList = addItem(myList, "orange");
myList = addItem(myList, "apple");
myList = addItem(myList, "milk");
cout << "List with three items:" << endl;
printList(myList);

myList = removeHead(myList);
cout << endl << "List with two items:" << endl;
printList(myList);
```

- Adds three items and Prints them
- Removes milk (last add)
- Print items (apple, orange)

Linked Lists In Practice

- With a library link lists are easy to use
- C++ stl Library provides a List container which can create lists of any data type
- Also one for dynamic arrays called a vector
- In practice the C++ containers are used since they are stable, efficient, and prevent memory leaks