Module 3

Linked List

Linear Data Structures

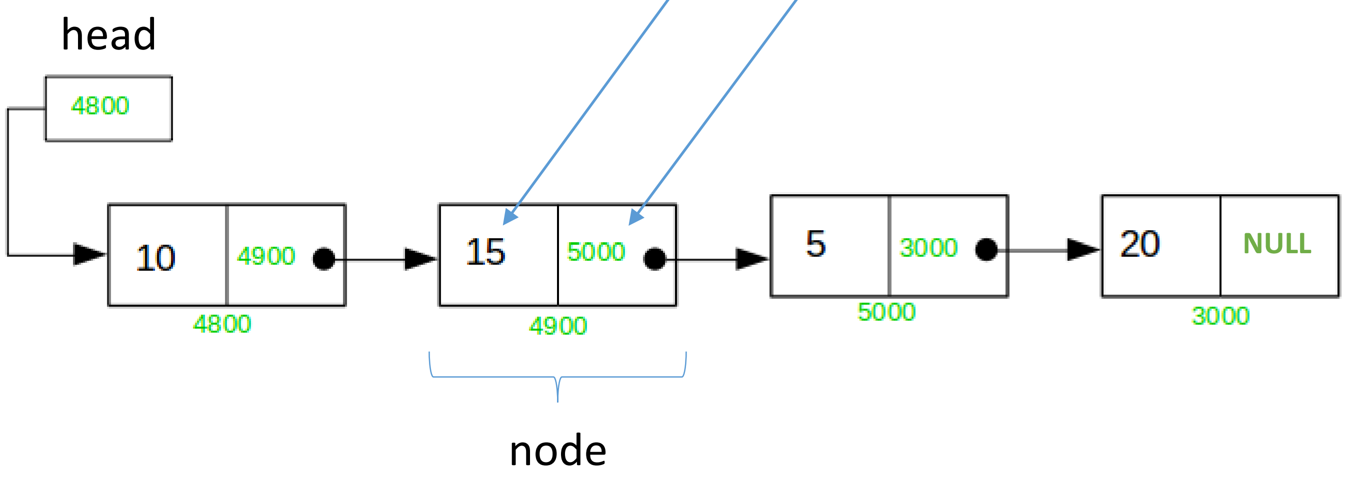
- **Linear Data Structures:** Data elements form a sequence or a linear list. The data is arranged in a linear fashion although the way they are stored in the memory need not to be sequential
  - Array
  - Linked List
  - Stack
  - Queue

Linked List

- A linked list is a linear data structure
  - In which the elements are not (necessarily) stored at contiguous memory locations
    - Separate memory allocation request is made for each element
    - Consecutive memory blocks may or may not be assigned
  - Each element (node) is divided into two memory blocks: data (key) value and reference to the next element
  - Stores the additional information in each element to find the sequence of elements in the list
  - Address of the first node or the head gives the access of the complete list

Linked List Representation

- Linked List: L
- Address of the first node: head L.*head*
- Information stored in node x key x.*key*
- Location (pointer) of next node next x.*next*



Basic Operations on Linked List

- Traversing the linked list
- Searching in the linked list
- Inserting a node to the linked list
- Deleting a node from the linked list

Traversing a Linked List

- Visiting each node of linked list L and printing the key values

```
LIST-TRAVERSE (L)
x = L.head
while x ≠ NULL
        Print x.key
        x = x.next
```

- Complexity: $\Theta(n)$

Searching in a Linked List

- To find the first node with key k in linked list L
  - Simple linear search
  - Returns a pointer to this node
  - If no object with key k appears in the list, then returns NULL

```
LIST-SEARCH(L, k)
x = L.head
while x ≠ NULL and x.key ≠ k
        x = x.next
return x
```

- Complexity: O(n) and $\Omega(1)$

Inserting into a Linked List

- Three different situations
  - Insertion at the beginning of the list
  - Insertion at the end of the list
  - Insertion at a specific position

- Insertion at the beginning of the list: Given a node x whose *key* attribute has already been set

  LIST-INSERT(L, x)
  1 x.*next* = L.*head*
  2 L.*head* = x

  Complexity: $\Theta(1)$

Inserting into a Linked List

- Insertion at the end of the list: Given a node x whose *key* attribute has already been set

```
LIST-INSERT(L, x)
p = L.head
while p.next ≠ NULL
        p = p.next
p.next = x
x.next = NULL
```

Complexity: $\Theta(n)$

Inserting into a Linked List

- Insertion at a given position: Given a node x whose *key* attribute has already been set. Insert the new node after k^{th} node

```
LIST-INSERT(L, x, k)
i = 1
p = L.head
while i ≠ k
        p = p.next
        i = i +1
x.next = p.next
p.next = x
```

Complexity: $O(n)$ and $\Theta(1)$

Deletion from a Linked List

- Removes a node x from a linked list L
  - Returns a pointer to x or returns the x.key

- Deletion from the beginning of the list: Delete the first node and save its key value

LIST-DELETE(L, k)
1 k = L.head.key
2 L.*head* = L.*head.next*

Complexity: $\Theta(1)$

Deletion from a Linked List

- Deletion from the end of the list: Delete the last node and save its key value

```
LIST-DELETE(L, k)
p = L.head
while p.next ≠ NULL
        q = p
        p = p.next
k = p.key
q.next = NULL
```

Complexity: $\Theta(n)$

Deletion from a Linked List

- Delete the j^{th} node and save its key value

```
LIST-DELETE(L, j, k)
i = 1
p = L.head
while i ≠ j
    q = p
    p = p.next
    i = i+1
k = p.key
if j == 1
    L.head = p.next
else
    q.next = p.next
```

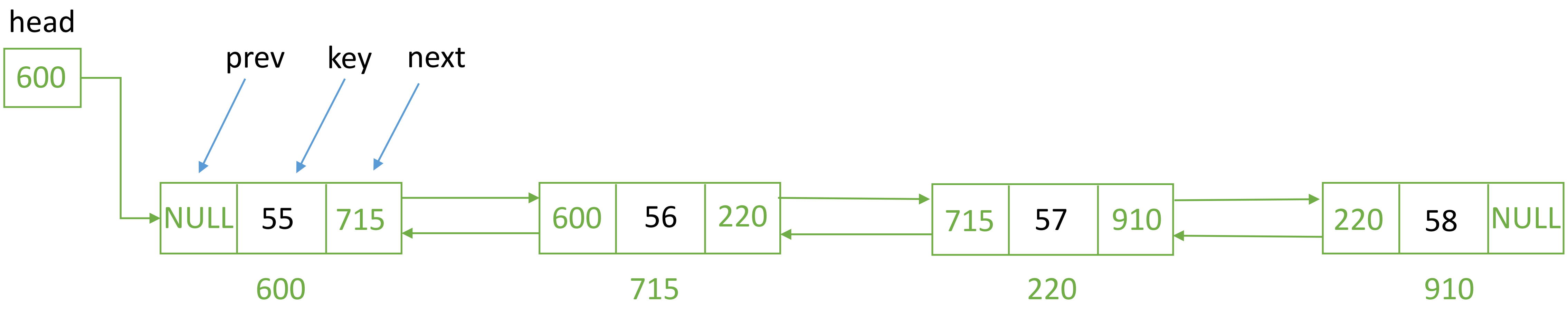
Complexity: $\mathrm{O}(n)$

Array vs. Linked List

- Linked list provides the following advantages over arrays
  1) Dynamic size → Better memory utilization
  2) Ease of insertion/deletion
- Linked lists have following drawbacks over arrays
  1) Random access is not allowed in the linked list. We have to access nodes sequentially starting from the first node
  2) Extra memory space for a pointer is required with each element of the list

Doubly Linked List

Doubly Linked List Representation



- prev: Location (pointer) to the previous node
- next: Location (pointer) to the next node
- key: Information

Doubly Linked List

- Advantages
  - Node will not only have the reference (pointer) to the next node but also to the previous node
    - Given a node, we can navigate in both directions
  - In doubly linked list, a node can be deleted even if we don’t have previous node’s address
    - In singly linked list, a node cannot be removed unless we have pointer to the predecessor
- Disadvantages
  - Each node requires an extra pointer, requiring more space
  - The insertion or deletion of a node takes a bit longer time (more pointer operations)

Basic Operations on Doubly Linked List

- Traversing the doubly linked list: Same as singly linked list
- Searching in the doubly linked list: Same as singly linked list
- Inserting a node to the doubly linked list
- Deleting a node from the doubly linked list

Inserting into a Doubly Linked List

- Three different situations
  - Insertion at the beginning
  - Insertion at the end
  - Insertion at a given position

- Insertion at the beginning of the list: Given a node x whose *key* attribute has already been set

LIST-INSERT(L, x)
1 x.*next* = L.*head*
2 *x.prev* = NULL
3 L.*head.prev* = x
4 L.*head* = x

Complexity: $\Theta(1)$

Inserting into a Doubly Linked list

- Insertion at the end of the list: Given a node x whose *key* attribute has already been set

```
LIST-INSERT(L, x)
p = L.head
while p.next ≠ NULL
        p = p.next
p.next = x
x.prev = p
x.next = NULL
```

Complexity: $\Theta(n)$

Inserting into a Doubly Linked List

- Insertion at a given position: Given a node x whose *key* attribute has already been set. Insert the new node after k^{th} node

```
LIST-INSERT(L, x, k)
i = 1
p = L.head
while i ≠ k
        p = p.next
        i = i +1
x.next = p.next
x.prev = p
p.next.prev = x
p.next = x
```

Complexity: $O(n)$

Deletion from a Doubly Linked List

- Removes a node x from a doubly linked list L
  - Returns a pointer to x or returns the x.key

- Deletion from the beginning of the list: Delete the first node and save its key value

LIST-DELETE(L, k)
1 k = L.head.key
2 L.*head* = *L.head.next*
3 L.*head.prev* = NULL

Complexity: $\Theta(1)$

Deletion from a Doubly Linked List

- Deletion from the end of the list: Delete the last node and save its key value

LIST-DELETE(L, k)
p = L.*head*
while p.*next* $\neq$ NULL
        p = p.*next*
k = p.*key*
p.*prev.next* = NULL

Complexity: $\Theta(n)$

Deletion from a Doubly Linked List

- Delete the j^{th} node and save its key value

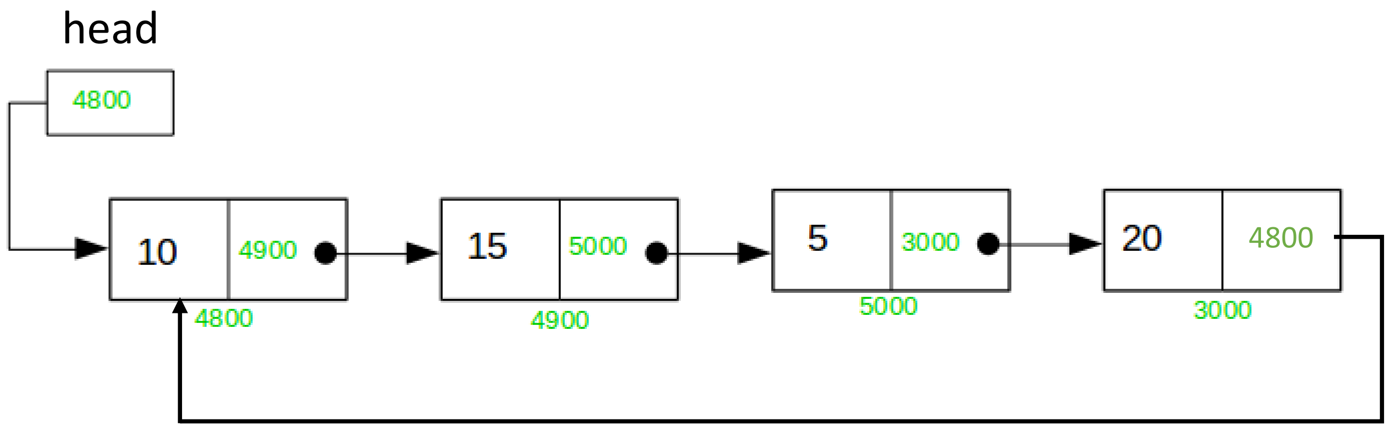
```
LIST-DELETE(L, j, k)
i = 1
p = L.head
while i ≠ j
    p = p.next
    i = i+1
k = p.key
p.next.prev = p.prev
p.prev.next = p.next
```

Complexity: $O(n)$

Circular Linked List

Circular Linked List

- All nodes are connected to form a circle
  - There is no NULL at the end: Last node points back to the first node



Traversing a Circular Linked List

- Visiting each node of linked list L and printing the key values

LIST-TRAVERSE (L)
x = L.*head*
do
    Print x.*key*
    x = x.*next*
while (x $\neq$ L.*head*)

- Complexity: $\Theta(n)$

Searching in a Circular Linked List

- To find the first node with key k in linked list L
  - Simple linear search
  - Returns a pointer to this node

If no object with key k appears in the list, then returns NULL

```
LIST-SEARCH (L, k)
x = L.head
do
    if (x.key == k)
        return x
    x = x.next
while (x ≠ L.head)
return NULL
```

- Complexity: O(n) and $\Omega(1)$

Inserting into a Circular Linked List

- Three different situations
  - Insertion at the beginning of the list
  - Insertion at the end of the list
  - Insertion at a specific position

- Insertion at the end of the list: Given a node x whose *key* attribute has already been set

```
LIST-INSERT (L, x)
p = L.head
while (p.next ≠ L.head)
    p = p.next
p.next = x
x.next = L.head
```

Complexity: $\Theta(n)$

Inserting into a Circular Linked List

- Insertion at the beginning of the list: Given a node x whose *key* attribute has already been set

```
LIST-INSERT (L, x)
p = L.head
while (p.next ≠ L.head)
    p = p.next
p.next = x
x.next = L.head
L.head = x
```

- Complexity: $\Theta(n)$

Inserting into a Circular Linked List

- Insertion at a given position: Given a node x whose *key* attribute has already been set. Insert the new node after k^{th} node

```
LIST-INSERT(L, x, k)
i = 1
p = L.head
while i ≠ k
        p = p.next
        i = i +1
x.next = p.next
p.next = x
```

Complexity: $O(n)$ and $\Theta(1)$

Deletion from a Circular Linked List

- Removes a node x from a doubly linked list L
  - Returns a pointer to x or returns the x.key

- Deletion from the beginning of the list: Delete the first node and save its key value

```
LIST-DELETE(L, k)
p = L.head
while p.next ≠ L.head
        p = p.next
k = p.key
L.head = L.head.next
p.next = L.head
```

Complexity: $\Theta(n)$

Deletion from a Circular Linked List

- Deletion from the end of the list: Delete the last node and save its key value

```
LIST-DELETE(L, k)
p = L.head
while p.next ≠ L.head
        q = p
        p = p.next
k = p.key
q.next = p.next
```

Complexity: $\Theta(n)$

Deletion from a Circular Linked List

- Delete the $\mathrm{j^{th}}$ node and save its key value

```
LIST-DELETE(L, j, k)
i = 1
p = L.head
while i ≠ j
    q = p
    p = p.next
    i = i+1
k = p.key
q.next = p.next
```

Complexity: $O(n)$

Reading Assignment

- Doubly Circular Linked List
  - Traversal
  - Search
  - Insertion
  - Deletion
- Applications of linked list in computer science
  - Implementation of stacks and queues
  - Implementation of graphs: Adjacency list representation
  - Manipulation of polynomials by storing constants and exponents in the node of linked list
  - Representing sparse matrices/tables

References

- Saymour L., **“Data Structures”,** Schaum’s Outline Series, McGraw Hill, Revised First Edition
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, **“Introduction to Algorithms”**, The MIT Press
- Sahni, S., **“Data Structures, Algorithms, and Applications in C++”,** WCB/McGraw-Hill