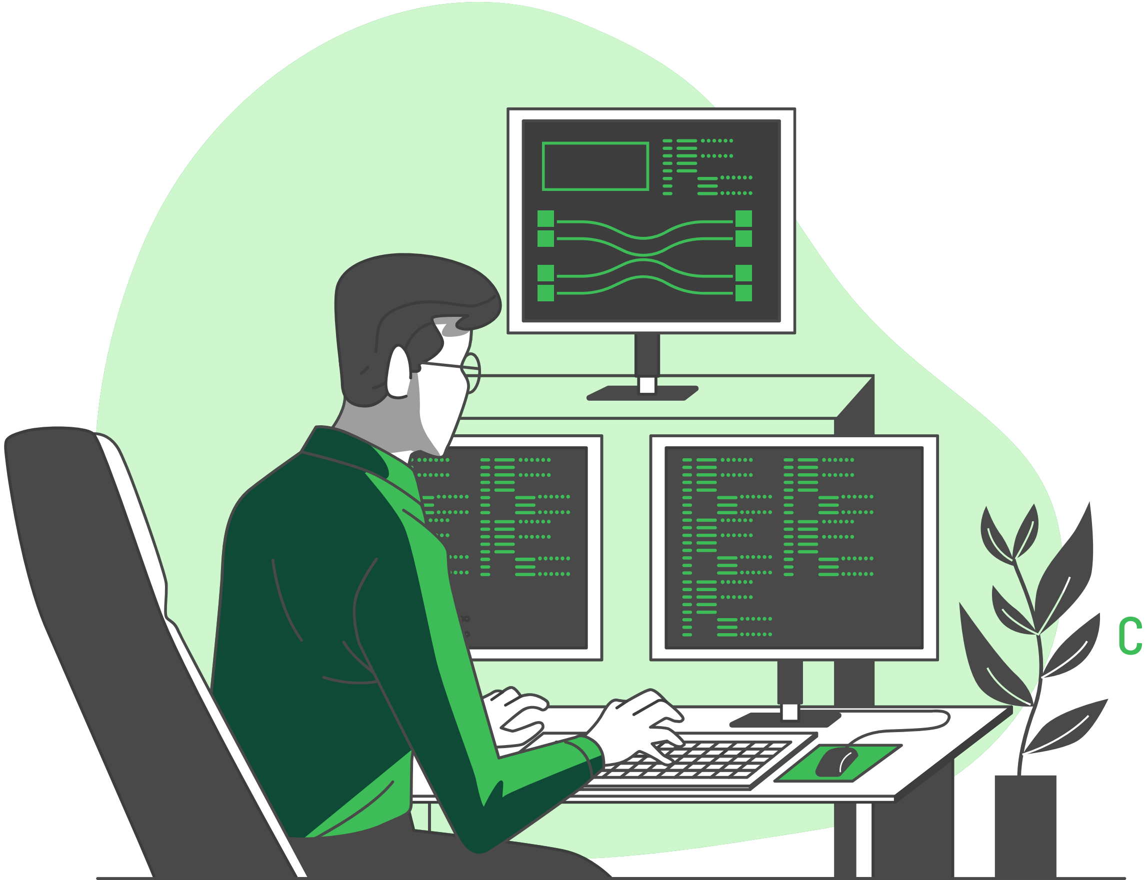
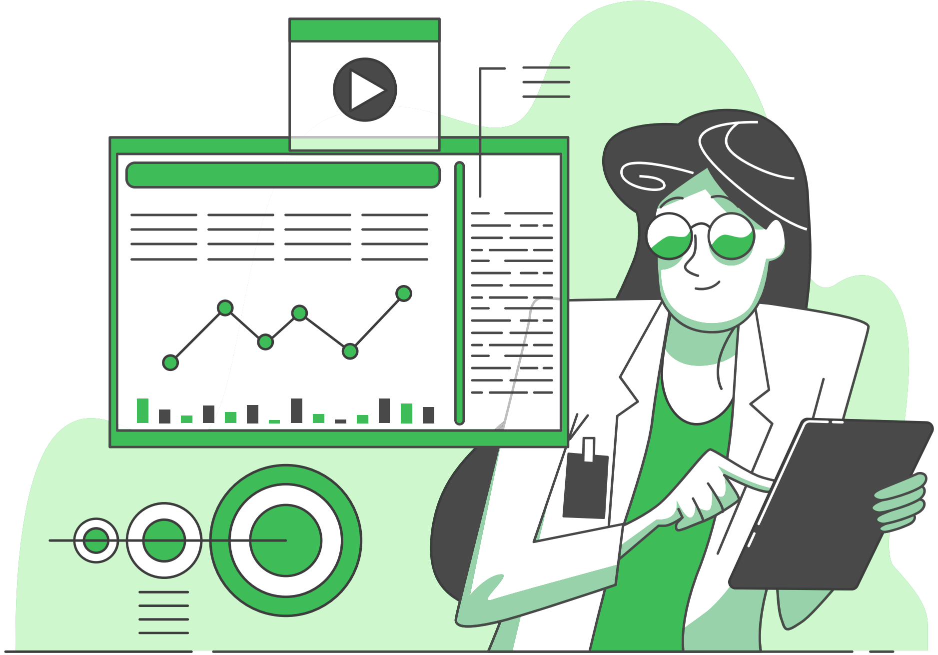


Linked List

CS 251 - Data Structures and Algorithms

Note:
Slides complement the discussion in class

Table of Contents



01

Singly Linked List

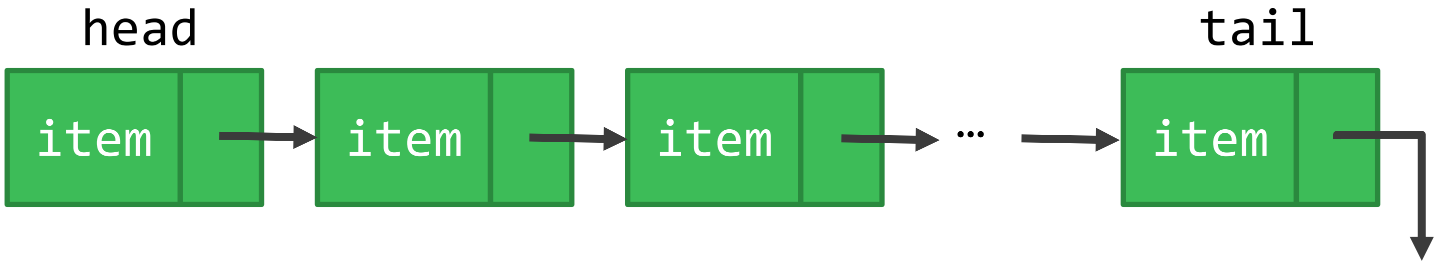
Dynamic linear sequence of items

Node

item: The content of the node. It could be as simple as a single value, or as complex as another data structure.
next: The address in memory to the next node in the list.

We use variables to store the address in memory of a node.

Singly Linked List



We keep the address in memory of the first node of the list. For simplicity, we call it **head** or **first**.

For certain implementations, it is convenient to keep the address in memory of the last node of the list. For simplicity, we call it **tail** or **last**.

We start an empty linked list by initializing the head address to **null** or **nil**.

Insertion at the Front

```
algorithm InsertAtFront(head:node, x:item) → node
   let newnode be a new Node
   newnode.item ← x

   if head is null then
      newnode.next ← null
      return newnode
   end if

   newnode.next ← head
   return newnode
end algorithm
```

The returned address in memory references to the new head of the singly linked list.

Populating an empty list at the front:

```
head ← null
head ← InsertAtFront(head, X)
head ← InsertAtFront(head, Y)
```

Insertion at the Back (ver. 1)

```
algorithm InsertAtBack(head:node, x:item) → node
   let newnode be a new Node
   newnode.item ← x
   newnode.next ← null

   if head is null then
      return newnode
   end if

   n ← head
   while n.next is not null do
      n ← n.next
   end while

   n.next ← newnode
   return head
end algorithm
```

The returned address in memory references to the (maybe new) head of the singly linked list.

Insertion at the Back (ver. 2)

```
algorithm InsertAtBack(tail:node, x:item) → node
   let newnode be a new Node
   newnode.item ← x
   newnode.next ← null

   if tail is null then
      return newnode
   end if

   tail.next ← newnode
   return newnode
end algorithm
```

The returned address in memory references to the new tail of the singly linked list.

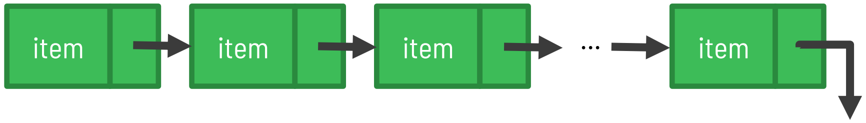
Search

```
algorithm Search(head:node, x:item) → node

   current ← head
   while current is not null do
      if current.item = x then
         return current
      end if
      current ← current.next
   end while

   return null
end algorithm
```

Delete

```
algorithm Delete(head:node, x:item) → node

   if head is null then
      return null
   end if

   if head.item = x then
      temp ← head
      head ← head.next
      delete temp
      return head
   end if

   prev ← null
   current ← head
   while current is not null and current.item ≠ x do
      prev ← current
      current ← current.next
   end while

   if current is not null then
      prev.next ← current.next
      delete current
   end if

   return head
end algorithm
```

Singly Linked List

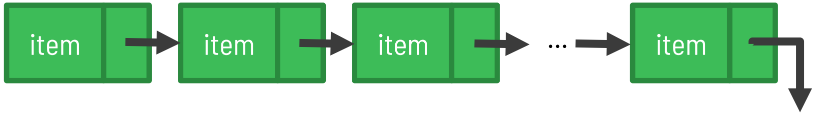


Q: How much space is required to keep track of the last node in a singly linked list?
One address: $\Theta(1)$

Q: How many operations are required to insert an item at the start/end of a singly linked list?
Start? New node + update addresses: $\Theta(1)$
End? Tracking last node?: $\Theta(1)$
End? Sorry, no tracking: $\Theta(n)$

Q: How many operations are required to insert an item somewhere in a singly linked list?
Find a location + update addresses:
$O(n) + O(1) \in O(n)$

Singly Linked List in Java




```java
public class Node<Item>
{
   // The item of the node
   public Item item;

   // The pointer to the next node
   // in the list
   public Node<Item> next;

   // Constructor of the class
   public Node(Item item)
   {
      this.item = item;
      next = null;
   }

   // ... more operations
}
```

```java
public class SinglyLinkedList<Item>
{
   // Pointer to the first node of the list
   private Node<Item> first;

   // Number of nodes of the list
   private int size;

   // Constructor of the class
   public SinglyLinkedList()
   {
      first = null;
      size = 0;
   }

   // Insert an item at the end of the list
   public void insert(Item item)
   {
      if (first == null)
      {
         first = new Node<Item>(item);
         size += 1;
         return;
      }

      Node<Item> n = first;

      while (n.next != null)
      {
         n = n.next;
      }

      n.next = new Node<Item>(item);
      size += 1;
   }

   // ... more operations
}
```

Arrays vs. Linked Lists

Array

Pros

Direct access to any element

No pointers (i.e., dynamic memory)

Linked List

Pros

Resizable

Easy insert at front or back of the list

Arrays vs. Linked Lists

Array

Cons

Fixed size

Enough consecutive space in memory

Linked List

Cons

Space overhead due to pointers

No direct access to internal nodes

Java Array Lists

Array Lists are not linked lists, they are arrays

Not allowed in programming projects unless stated in the description file

02
Doubly Linked List

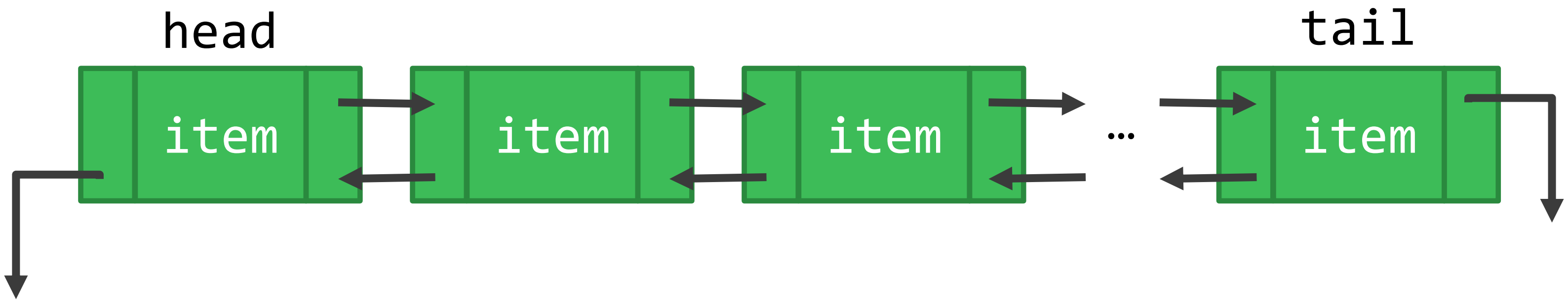
Two memory addresses per node

Doubly Node

item: The content of the node. It could be as simple as a single value, or as complex as another data structure.
prev: The address in memory to the previous node in the list.
next: The address in memory to the next node in the list.

We use variables to store the address in memory of a node.

Doubly Linked List



We keep the address in memory of the first node of the list. For simplicity, we call it **head** or **first**.

For certain implementations, it is convenient to keep the address in memory of the last node of the list. For simplicity, we call it **tail** or **last**.

We start an empty doubly list by initializing the head address to **null, nil**, or **none**.

Insertion at the front (no tail tracking)

```
algorithm InsertFront(head:node, x:item) → node
   let newnode be a new Node
   newnode.item ← x
   newnode.prev ← null
   newnode.next ← head

   if head is not null then
      head.prev ← newnode
   end if

   return newnode
end algorithm
```

The returned address in memory references to the new head of the doubly linked list.

Populating an empty list at the front:

```
head ← null
head ← InsertFront(head, X)
head ← InsertFront(head, Y)
```

Insertion at the back (ver. 1) (no tail tracking)

```
algorithm InsertBack(head:node, x:item) → node
   let newnode be a new Node
   newnode.item ← x
   newnode.next ← null

   if head is null then
      newnode.prev ← null
      return newnode
   end if

   n ← head
   while n.next is not null do
      n ← n.next
   end while

   n.next ← newnode
   newnode.prev ← n
   return head
end algorithm
```

The returned address in memory references to the (maybe new) head of the doubly linked list.

Insertion at the back (ver. 2) (tail tracking)

```
algorithm InsertBack(tail:node, x:item) → node
   let newnode be a new Node
   newnode.item ← x
   newnode.prev ← tail
   newnode.next ← null

   if tail is not null then
      tail.next ← newnode
   end if

   return newnode
end algorithm
```

The returned address in memory references to the new tail of the doubly linked list.

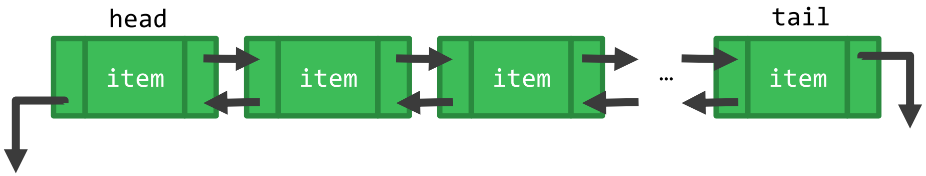
Search an item

```
algorithm Search(head:node, x:item) → node

   current ← head
   while current is not null do
      if current.item = x then
         return current
      end if
      current ← current.next
   end while

   return null
end algorithm
```

Delete a node (tail tracking)

```
algorithm Delete(h:node, t:node, x:node) → node

   if x is null then
      return
   end if

   if x.prev is not null then
      x.prev.next ← x.next
   else
      head ← x.next
   else if

   if x.next is not null then
      x.next.prev ← x.prev
   else
      tail ← x.prev
   end if

   x.prev ← null
   x.next ← null
   delete x

end algorithm
```

Doubly Linked List



Q: How much space is required to keep track of the last node in a doubly linked list?
One address: $\Theta(1)$

Q: How many operations are required to insert an item at the front/back of a doubly linked list?
Start? New node + update addresses: $\Theta(1)$
End? Tracking tail?: $\Theta(1)$
End? Sorry, no tracking: $\Theta(n)$

Q: How many operations are required to insert an item somewhere in a doubly linked list?
Find a location + update addresses:
$O(n) + \Theta(1) \in O(n)$

EOF

Do you have any questions?

CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, infographics & images by Freepik and illustrations by Stories