Semantics

¬, ∃x,∀y (Highest, do first)

∧

∨

→ (Lowest, do last)

Basic Equivalences

Negation
¬¬A

Basic Equivalences

Some Conversions
$A \to B \equiv \neg A \vee B$
$\neg(A \to B) \equiv A \wedge \neg B$
$A \to B \equiv A \wedge \neg B \to$ False
∧ and ∨ are associative
$(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$
$(A \vee B) \vee C \equiv A \vee (B \vee C)$
∧ and ∨ are commutativity
$A \wedge B \equiv B \wedge A$
$A \vee B \equiv B \vee A$
∧ and ∨ are Distributivity
$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$

Proof Rules

Conjunction (Conj)	A, B / $A \wedge B$	
Simplification (Simp)	$A \wedge B$ / A and	$A \wedge B$ / B
Addition (Add)	A / ☐ ☐∨B and	B / ☐ ☐∨B
Disjunctive Syllogism (DS)	A ∨ ☐ ☐, $\neg A$ / B	A ∨ ☐ ☐, $\neg A$ / B
Modus Ponens (MP)	$A, A \to B$ / B	
Conditional Proof (CP)	From$A$, derive ☐ ☐/$A \to B$	
Double Negation (DN)	$\neg\neg A$ / A	A /$\neg\neg A$

Proof Rules (cont)

Contradiction (Contr)	A, $\neg A$ / False
Indirect Proof (IP)	From$\neg A$, derive False / A

These are all fractions with the first term appearing on top and the second one on the bottom.

/ this slash denotes where a fraction will be located

CS 310 Lecture 5

Array Data Structure	A linear data structure defined as a collection of elements with the same or different data types.
	They exist in both single and multiple dimensions

CS 310 Lecture 5 (cont)

Terms to understand the concept of Array.	o Element − Each item stored in an array is called an element.
	o Index − Each location of an element in an array has a numerical index.

CS 310 Lecture 5

Array Update Operation

1. Start
2. Set LA[K-1] = ITEM
3. Stop

CS 310 Lecture 5

Common Features of Linked List

CS 310 Lecture 5 (cont)

➢Node: Each element in a LL is represented by a node, contains two components:	➢Data: The actual data or value associated with the element.	➢Next Pointer: A reference or pointer or address to the next node in the LL.

➢Head: The first node in a LL is called the “head.” It serves as the starting point.

➢Tail: The last node in a linked list is called the “tail.”

➢Data structures can be added to or removed from the LL during execution.

➢Unlike an array, LL is a dynamically allocated DS that can grow and shrink.

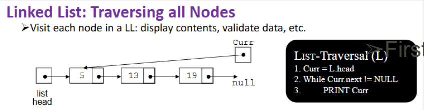
➢No elements need to be shifted after insertion and deletion.

CS 310 Lecture 5 (cont)

➢Various DSs can be implemented using an LL, such as stack, queue, graphs, hash, etc.

➢Linked list contains 0 or more nodes. Last node points to null(address 0)

CS 310 Lecture 5



LIST-Traversal (L)
1. Curr = L.head
2. While Curr.next != NULL
3. PRINT Curr

CS 310 Lecture 5

Linked List: Searching a Node

LIST-Searching (L,k)

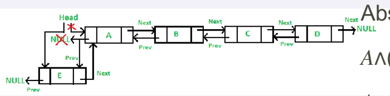
1. Curr = L.head

2. While Curr != NULL and Curr.key != k

3. Curr = Curr.next

4. return Curr

CS 310 Lecture 5



Doubly-linked list: Inserting at the beginning

➢The task can be performed by using the following 5 steps:

➢Firstly, allocate a new node.

➢Now put the required data in the new node.

➢Make the next of new_node point to the current head of the DLL.

➢Make the previous of the current head point to new_node.

➢Lastly, point head to new_node.

Basic Equivalences

Disjunction

A v True ≣ True

A v False ≣ A

A v A ≣ A

A v ¬A ≣ True

Basic Equivalences

Absorption Laws

$A \wedge (A \vee B) \equiv A$

$A \vee (A \wedge B) \equiv A$

$A \wedge (\neg A \vee B) \equiv A \wedge B$

$A \vee (\neg A \wedge B) \equiv A \vee B$

Program Correctness

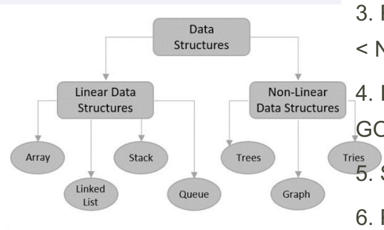
AA (Assignment axiom)	$\{Q(x/t)\}\ x := t\ \{Q\}$	
Consequence rules (A & B)	$P \rightarrow R$ and $\{R\}$ S $\{Q\}$ / $\{P\}$ ☐ ☐ $\{Q\}$	$\{P\}$ S $\{T\}$ and ☐ ☐ $\rightarrow Q$ / $\{P\}$ S $\{Q\}$

Loop invariants: A loop invariant is a condition that does not change after a loop has executed I.e. P

Derived Proof Rules

Modus Tollens (MT)	$A \to B, \neg B$ / $\neg A$
Hypothetical Syllogism (HS)	$A \to B, B \to C$ / $A \to C$
Proof by Cases (Cases)	$A \vee B, A \to C, \square \to C$ / C
Constructive Dilemma (CD)	$A \vee B, A \to C, B \to D$ / $C \vee D$
Destructive Dilemma (DD)	$A \to B, C \to D, \neg B \vee \neg D$ / $\neg A \vee \neg C$

CS 310 Lecture 5



CS 310 Lecture 5

Operations in Arrays

o Traverse − print all the array elements one by one.

o Insertion − Adds an element at the given index.

o Deletion − Deletes an element at the given index.

o Search − Searches an element using the index or value.

o Update − Updates an element at the given index.

o Display − Displays the contents of the array.

CS 310 Lecture 5

Array Search Operation

1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] == ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop

CS 310 Lecture 5

➢Singly Linked List: Every node stores the address of the next node in the list and the last node has the next address NULL.

CS 310 Lecture 5

Linked List: Operations

➢Accessing Elements/Traversing: Accessing a specific element in a linked list takes O(n) time since nodes are stored in non-contiguous locations, so random access is not possible.

➢Searching: Searching a node in an LL takes O(n) time, as the whole list needs to be traversed in the worst case.

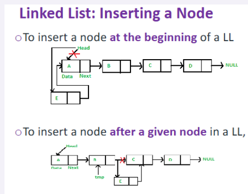
➢Insertion: If we are at the position where we insert the element, insertion takes O (1) time.

➢Deletion a/Destroy the list: Deletion takes O(1) time if we know the element's position to be deleted.

CS 310 Lecture 5

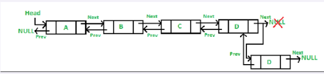
➢Doubly Linked Lists: Each node has two pointers: one pointing to the next node and one
pointing to the previous node. Allows for efficient traversal in both directions.

CS 310 Lecture 5



LIST-Insert (L,x,k)
1. if L.Head == NULL
2. L.Head = x and Exit
3. While Curr.key !=k and Curr !=NULL
4. prevN = Curr
5. Curr = Curr.Next
6. If PrevN == NULL
7. x.next = L.Head
8. Head = x and exit
9. PrevN = Curr and Curr = Curr.Next
10. x.Next = Curr
11. PrevN.Next = x

CS 310 Lecture 5



Doubly-linked list: Inserting at the end

➢This can be done using the following 7 steps:

➢Create a new node (say new_node).

➢Put the value in the new node.

➢Make the next pointer of new_node as null.

➢If the list is empty, make new_node as the head.

➢Otherwise, travel to the end of the linked list.

➢Now make the next pointer of last node point to new_node.

➢Change the previous pointer of new_node to the last node of the list.

Basic Equivalences

Conjunction
A ∧ True ≡ A
A ∧ False ≡ False
A ∧ A ≡ A
A ∧ ¬A ≡ False

Basic Equivalences

Absorption Laws
$A \wedge (A \vee B) \equiv A$
$A \vee (A \wedge B) \equiv A$
$A \wedge (\neg A \vee B) \equiv A \wedge B$
$A \vee (\neg A \wedge B) \equiv A \vee B$

Program Correctness

AA (Assignment axiom)	$\{Q(x/t)\}\ x \coloneqq t\ \{Q\}$	
Consequence rules (A & B)	$P \rightarrow R$ and $\{R\}\ S\ \{Q\}$ / $\{P\}$ ☐ ☐ $\{Q\}$	$\{P\}\ S\ \{T\}$ and ☐ ☐ $\rightarrow Q$ / $\{P\}\ S\ \{Q\}$

Program Correctness (cont)

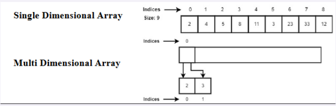
Composition rule	$\{P\}\ S1\ \{Q\}$ and $\{Q\}\ S2\ \{R\}$ / $\{P\}$ ☐ ☐1;$S2$ $\{R\}$	
If-then Rule	$\{P \wedge C\}\ S\ \{Q\}$ and ☐ ☐ $\wedge \neg C \rightarrow Q$	$\{P\}$ if C then S $\{Q\}$

Program Correctness (cont)

If-then-else rule	$\{P \wedge C\}\ S1\ \{Q\}$ and $\{P \wedge \neg C\}\ S2\ \{Q\}$ / $\{P\}$ if C then $S1$ else ☐ ☐2 $\{Q\}$
While rule	$\{P \wedge C\}\ S\ \{P\}$ / $\{P\}$ while C do S $\{P \wedge \neg C\}$

Loop invariants: A loop invariant is a condition that does not change after a loop has executed I.e. P

CS 310 Lecture 5



CS 310 Lecture 5

Array Deletion Operation

1. Start
2. Set J = K-1
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop

N - is the size of the array

CS 310 Lecture 5

Array Insertion Operation

1. Start
2. Create an Array of a desired datatype and size.
3.Initialize a variable 'i' as 0.
4. Enter the element at the i-th index of the array.
5. Increment i by 1
6. Repeat Steps 4 & 5 until the end of the array.

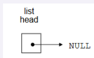
CS 310 Lecture 5 (cont)

7. Stop

CS 310 Lecture 5

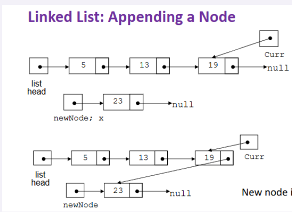
≻Circular Linked Lists: A circular linked list is a type of linked list in which the first and the last
nodes are also connected to form a circle. There is no NULL at the end.

CS 310 Lecture 5



Linked List: Empty List
≻If a list currently contains 0 nodes, it is called the empty list.
≻In this case, the list head points to null

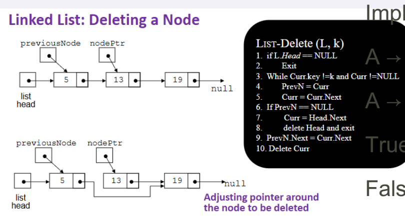
CS 310 Lecture 5



LIST-Append (L; x)
1. if L.head == NULL
2. L.head = x and Exit
3. Curr = L.head
3. While Curr.next != NULL
Curr = Curr.next
4. Curr.next = x

New Node is added to the end of the list

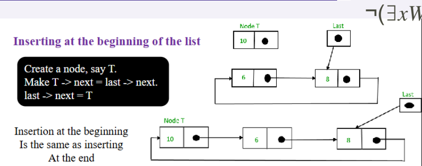
CS 310 Lecture 5



Adjusting pointer around the node to be deleted

LIST-Delete (L, k)
1. if L.Head == NULL
2. Exit
3. While Curr.key !=k and Curr !=NULL
4. PrevN = Curr
5. Curr = Curr.Next
6. If PrevN == NULL
7. Curr = Head.Next
8. delete Head and exit
9. PrevN.Next = Curr.Next
10. Delete Curr

CS 310 Lecture 5



Circular-linked list operations:
≻Insertion: Inserting At the Beginning, at the end, and after a given node.
≻Deletion: Deleting from the Beginning, the end, and a Specific Node
≻Display: This process displays the elements of a CLL.

Basic Equivalences

Implication

A → True ≡ True

A → False ≡ ¬A

True → A ≡ A

False → A ≡ True

A → A ≡ True

Basic Equivalences

De Morgan's Laws

$\neg(A \wedge B) \equiv \neg A \vee \neg B$

$\neg(A \vee B) \equiv \neg A \wedge \neg B$

Quantifiers

"An equivalence to be careful with"

$\exists x(p(x) \rightarrow q(x)) \equiv \forall x p(x) \rightarrow \exists x q(x)$

Quantifiers

Negations of quantifiers

$\neg(\forall x W) \equiv \exists x \neg W$

$\neg(\exists x W) \equiv \forall x \neg W$

Quantifiers

Formalize English sentences and entire arguments into FOPC
$\forall x$ quantifies a conditional
$\exists x$ quantifies a conjunction
$\forall x$ with conditional for “all,” “every,” and “only.”
$\exists x$ with conjunction for “some,” “there is,” and “not all.”
$\forall x$ with conditional or $\neg\exists x$ with conjunction for “no A is B.”
$\exists x$ with conjunction or $\neg\forall x$ with conditional for “not all A’s are B.”

Inference Rules FOPC

UI	Universal instantiation requires that t is free to replace x in $W(x)$:	$\forall x W(x)$ / ☐(t)	There are two special cases for UI:	$\forall x W(x)$ / ☐(x)	$\forall x W(x)$ / ☐(c)

Inference Rules FOPC (cont)

EI	Existential generalization requires that t is free to replace x in $W(x)$	$W(t)$ / $\exists x W(x)$

Inference Rules FOPC (cont)

EG	Existential generalization requires that ☐ ☐ is free to replace x in $W(x)$:	$W(t)$ / $\exists x W(x)$	There are two special cases for EG:

Inference Rules FOPC (cont)

UG	Universal Generalization	$W(x)$ / $\exists x W(x)$	$W(c)$ / $\exists x W(x)$	$\exists x W(x)$ / $W(c)$
UI & EI Add A and E from problem; UG, EG, Take the away A and E in the problem.				

CS 310 Lecture 5

Array Traversal Operation
1. Start
2. Initialize an Array, LA. // 1. Initialize an array called LA
3. Initialize, i = 0. // 2. Set i - 0
4. Print the LA[i] and increment i. // 3. Repeat Steps 4-5 while i < N
5. Repeat Step 4 until the end of the array. // 4. Print LA[i]

CS 310 Lecture 5 (cont)

6. End // 5. Increment the value of i by one. (Set i = i + 1)

// represent possible modifications you can do that would still be counted as correct

CS 310 Lecture 5

≻An abstract data type (ADT) in data structure is a data type defined with the help of some attributes and some functions

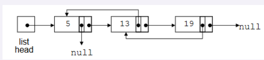
≻An abstract data type in the data structure can be	≻A list data structure	≻A stack data structure	≻A queue data structure

CS 310 Lecture 5 (cont)

≻Linked List: A LL is a linear data structure constructed like a chain of nodes where	≻Each node contains a data field	≻A reference(link/address/array-Indices) to the next node in the list.

≻Unlike Arrays, Linked List elements are not stored at a contiguous location.

CS 310 Lecture 5



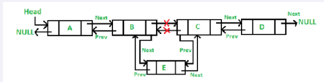
Doubly-linked list: Operations

≻Insertion: Inserting At the Beginning, at the end, after a given node, and before a given node.

≻Deletion: Deleting from the Beginning, end, and a specific node of the list

≻Display: This process displays the elements of a doubly LL.

CS 310 Lecture 5



Doubly-linked list: Inserting after a given node

≻ Inserting after a given node can be done by:

≻Firstly create a new node (say new_node).

≻Now insert the data in the new node.

≻Point the next of new_node to the next of prev_node.

≻Point the next of prev_node to new_node.

≻Point the previous of new_node to prev_node.

≻Change the pointer of the new node's previous pointer to new_node.