CSCE 2100: Computing Foundations 1

The List Data Model

Tamara Schneider
Jorge Reyes-Silveyra
Fall 2012

Terminology [1]

- A **list** is a finite set of 0 or more elements
- All elements in the list are most of the time of the same type T
- The elements of a list are separated by commas: $(a_1, a_2, \ldots a_n)$
  - Exception: A string as in a list of characters may be represented without commas
- Duplicate elements are generally allowed

2

Review of Terminology [2]

- **Length** of a list: number of elements in the list
  - The empty list is represented by () or ε
- The first list element is called **head**
  - The head is a single list element!
- The remainder of the list is called **tail**
  - The tail is a list

3

Review of Terminology [3]

- **Sublist**: contiguous part of the list from position $i \geq 1$ to position $j \leq n$
- **Subsequence:** Subset of the elements of a list preserving the order of their occurrence in the original list
- **Prefix:** Sublist starting at the beginning of the list ($i = 1$)
- **Suffix:** Sublist terminating at the end of the list ($j = n$)

4

Example

List of integers: (4, 6, 2, 5, 2, 8, 3)

- **Length** of the list: 7
- The **head** of the list is 4
- The **tail** of the list is (6, 2, 5, 2, 8, 3)
- (6, 2, 5) and (4, 6, 2, 5) are **sublists**
- The tail is a **sublist**
- (4, 6, 5, 8) and (2, 2, 3) are **subsequences**
- (4, 6, 2, 5) and (4, 6) are **prefixes**
- (2, 8, 3) and (5, 2, 8, 3) are **suffixes**

5

List Operations [1]

Dictionary Operations

- Insertion: Insert an element x anywhere in the list.
  - If x is the new head, it is "pushed" onto the list resulting in $(x, a_1, a_2, \ldots a_n)$
- Deletion: Delete **one** occurrence of x
  - If x is the head: "pop the list"
- Search / Lookup: return TRUE if element is in the list, FALSE otherwise

6

List Operations [2]

- **Concatenation**: concatenating two lists
  $L = (a_1, a_2, \ldots a_n)$ and $M = (b_1, b_2, \ldots b_n)$ yields LM =
  $(a_1, a_2, \ldots a_n, b_1, b_2, \ldots b_n)$
  – For **the empty list** ε:
  $L\,\varepsilon = L = \varepsilon L$
- **first(L), last(L)** return first or last element of the list
- **retrieve (i, L)** returns element at position I
- **length(L)** returns the length of the list
- **isEmpty(L)** returns TRUE if the list is empty

7

Linked List Data Structure

- In C we can implement a linked list using a `struct`
- In C++ we can implement a linked list using 2 classes: `CNode` and `CLinkedList`.

```
/// \file node.h
class CNode{
  friend class CLinkedList;
  private:
    int m_nData;
    CNode* m_pNext;
  public:
    CNode(int data);
}; //CNode
```

8

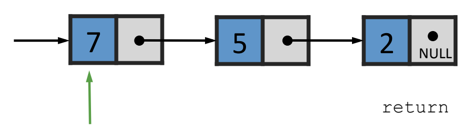
```
/// \file linkedlist.h
#include "node.h"
class CLinkedList{
  private:
    CNode* m_pHead;
    int m_nSize;
  public:
    CLinkedList();
    void addNode(int data);
    void removeNode(int data);
    bool searchNode(int data);
    void printList();
}; //CLinkedList
```

9

Linked List: Search

Check each element in the list until the search key has been found or the end of the list has been reached.

`search(2);`



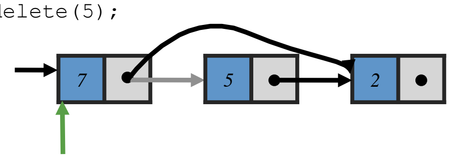
`return true;`

10

Linked List: Deletion

Check each element in the list until the search key has been found or the end of the list has been reached. If the element is found redirect the pointer of the previous element.

`delete(5);`

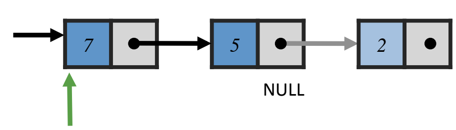


11

Linked List: Insertion

Duplicate elements may or may not be allowed! Find the end of the list. Add a new element.

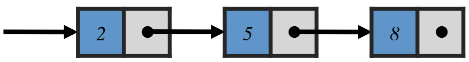
`insert(2);`



12

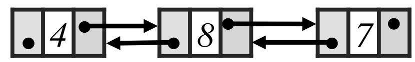
Sorted Lists (Represent Dictionaries)

- Elements are maintained in sorted order
- Insertion: Do not insert at the end, but at the appropriate space
- Deletion: same as "regular" lists
- Search: In average faster; why?



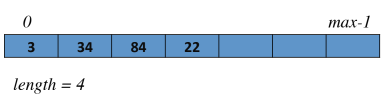
Doubly Linked Lists

- Each element contains a "previous" pointer and a "next" pointer.
- When inserting or deleting both pointers must be updated.



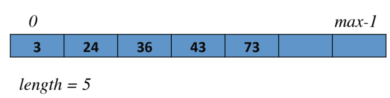
Array-Based List Implementation

- Create an array of size `MAX` to keep the list elements
- Introduce a variable `length` that keeps track of the number of elements in the list



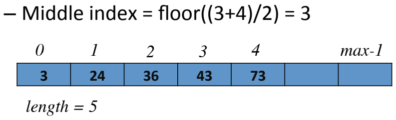
Sorted Array-Based Lists [1]

- The elements in the list are sorted
- How can we use this to improve the speed of `search(x)`?

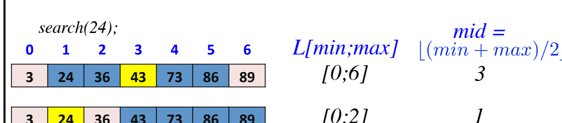


Sorted Array-Based Lists [2]

- Observation: The left half of the list contains smaller elements than the right half of the list
- Assume we are searching for x = 43.
  - Middle index of list [0;4] = 2 with element 36
  - Is x < 36? No, so search sublist L[3;4]
  - Middle index = floor((3+4)/2) = 3



Sorted Array-Based Lists [3]



- Calculate the mid element of the list
- If the element has been found, return "true"
- Otherwise, evaluate left or right side of list
  - Left side if x < L[mid]
  - Right side if x > L[mid]

Sorted Array-Based Lists [4]

- How long does the search take on an array-based sorted list?(Running time?)
- What needs to be done to insert elements into the list?
- How can we delete elements from the list?

Sorted Array-Based Lists [5]

```
bool bsrch(int x, int L[], int lo, int hi){
  int mid; //middle element of list
  if(lo > hi)return false; else{
    mid = (lo + hi)/2;
    if(x < L[mid])
      return bsrch(x, L, lo, mid-1);
    else if(x > L[mid])
      return bsrch(x, L, mid+1, hi);
    else return true; //L[mid] == x
  } //else
} //bsearch
```


Stacks

- Abstract data type based on list data model
- LIFO (last-in first-out)
- Stack operations
  - `push(x)` puts the element x on top of the stack
    `push(x)` onto (a_1, a_2, ... a_n) yields (a_1, a_2, ... a_n, x)
  - `pop()` removes the topmost element from stack
    `pop()` from (a_1, a_2, ... a_n) yields (a_1, a_2, ... a_{n-1})

Stack Example - Postfix Expressions [1]

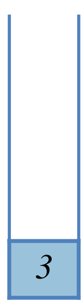
- Many compilers turn infix expression into postfix expressions.
- Then the postfix expressions can be evaluated via stacks.
  - Reading argument: push onto stack.
  - Reading operator: pop 2 elements from stack and evaluate. Push result onto stack.

Stack Example - Postfix Expressions [2]

- Infix expression: (3 + 4) * 2
- Convert to postfix:

3 *4* + *2* *

SYMBOL PROCESSED	STACK	ACTIONS
initial	ϵ	
3	3	*push* 3
4	3, 4	*push* 4
+	ϵ	*pop* 4; *pop* 3
		compute $7 = 3 + 4$
	7	*push* 7
2	7, 2	*push* 2
×	ϵ	*pop* 2; *pop* 7
		compute $14 = 7 \times 2$
	14	*push* 14

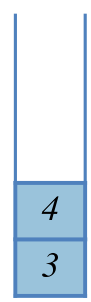


Stack Example - Postfix Expressions [2]

- Infix expression: (3 + 4) * 2
- Convert to postfix:

3 *4* + *2* *

SYMBOL PROCESSED	STACK	ACTIONS
initial	ϵ	
3	3	*push* 3
4	3, 4	*push* 4
+	ϵ	*pop* 4; *pop* 3
		compute $7 = 3 + 4$
	7	*push* 7
2	7, 2	*push* 2
×	ϵ	*pop* 2; *pop* 7
		compute $14 = 7 \times 2$
	14	*push* 14

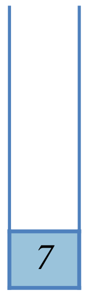


Stack Example - Postfix Expressions [2]

– Infix expression: (3 + 4) * 2
– Convert to postfix:

3 4 **[+]** *2* *

Symbol Processed	Stack	Actions
initial	ε	
3	3	*push* 3
4	3, 4	*push* 4
+	ε	*pop* 4; *pop* 3
		compute 7 = 3 + 4
	7	*push* 7
2	7, 2	*push* 2
×	ε	*pop* 2; *pop* 7
		compute 14 = 7 × 2
	14	*push* 14



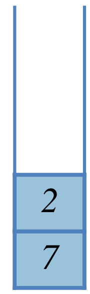
25

Stack Example - Postfix Expressions [2]

– Infix expression: (3 + 4) * 2
– Convert to postfix:

3 4 + **[2]** *

Symbol Processed	Stack	Actions
initial	ε	
3	3	*push* 3
4	3, 4	*push* 4
+	ε	*pop* 4; *pop* 3
		compute 7 = 3 + 4
	7	*push* 7
2	7, 2	*push* 2
×	ε	*pop* 2; *pop* 7
		compute 14 = 7 × 2
	14	*push* 14



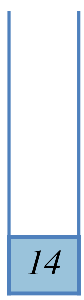
26

Stack Example - Postfix Expressions [2]

– Infix expression: (3 + 4) * 2
– Convert to postfix:

3 4 + *2* **[*]**

Symbol Processed	Stack	Actions
initial	ε	
3	3	*push* 3
4	3, 4	*push* 4
+	ε	*pop* 4; *pop* 3
		compute 7 = 3 + 4
	7	*push* 7
2	7, 2	*push* 2
×	ε	*pop* 2; *pop* 7
		compute 14 = 7 × 2
	14	*push* 14



27

Stack Operations

- `push(x)`
- `pop()`
- `clear()`
  Initializes stack to ensure that it is empty
- `isFull()`
  Although in theory the stack can grow infinitely, a stack implementation can only hold only a certain number of elements

28

Stack Implementation

<u>Option 1:</u> Use arrays.

<u>Option 2:</u> Use an implementation similar to linked lists with stack elements instead of list nodes. Since a linked list does not have a size limit, `isFull()` can always return `false`.

29

```
/// \file stack.h
#include "node.h"
class CStack{
  private:
    CNode* m_pTop;
    int m_nSize;
  public:
    CStack();
    void push(int data);
    int pop();
    bool isFull();
    bool isEmpty();
    void clear();
}; //CStack
```

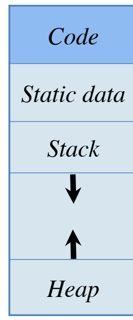
30

Stacks in Memory Allocation

- What happens if a function is called recursively? How do we distinguish between the different occurrences of variables with the same name?
- Each execution of a function is called an **activation**.
  - Associated objects are stored in activation record (parameters, return value, return address, local variables)

31

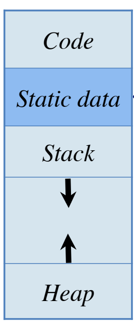
How is Runtime Memory Organized? [1]



Text segment. The compiled code of your program in the form of machine code.

32

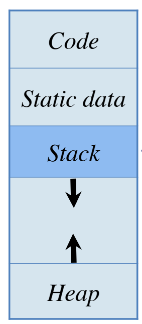
How is Runtime Memory Organized? [2]



Fixed size static data. Values of certain constants and external variables used by the program.

33

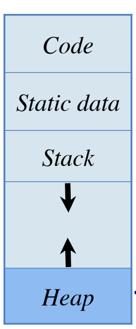
How is Runtime Memory Organized? [3]



Activation records for all currently live activations. Records are pushed onto the stack. A returning function pops the record. Parameters are also stored on the stack.

34

How is Runtime Memory Organized? [5]



Dynamically allocated objects (using `malloc`, `new`, etc)

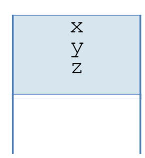
e.g. `str = malloc(20);`
`int y;`

If no place in the heap with sufficient space is found, heap size is increased

35

Example 1 - Multiple Functions [1]

```
void main(){
  int x,y,z;
  P();
}
void P(){
  int p1,p2;
  Q();
}
void Q(){
  int q1,q2,q3;
}
```



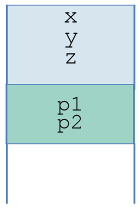
`main()` starts executing: Its activation space contains space for variables `x`, `y`, and `z`.

36

Example 1 - Multiple Functions [2]

```
void main(){
  int x,y,z;
  P();
}
void P(){
  int p1,p2;
  Q();
}
void Q(){
  int q1,q2,q3;
}
```

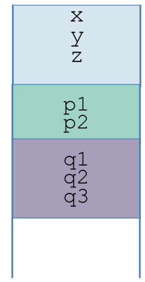
Activation record for `P` is pushed onto the stack.



Example 1 - Multiple Functions [3]

```
void main(){
  int x,y,z;
  P();
}
void P(){
  int p1,p2;
  Q();
}
void Q(){
  int q1,q2,q3;
}
```

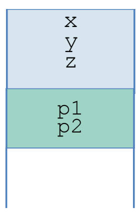
Activation record for `Q` is pushed onto the stack.



Example 1 - Multiple Functions [4]

```
void main(){
  int x,y,z;
  P();
}
void P(){
  int p1,p2;
  Q();
}
void Q(){
  int q1,q2,q3;
}
```

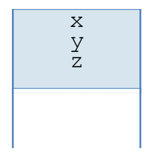
`Q` returns and its activation record is popped off the stack.



Example 1 - Multiple Functions [5]

```
void main(){
  int x,y,z;
  P();
}
void P(){
  int p1,p2;
  Q();
}
void Q(){
  int q1,q2,q3;
}
```

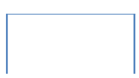
`P` returns and its activation record is popped off the stack.



Example 1 - Multiple Functions [6]

```
void main(){
  int x,y,z;
  P();
}
void P(){
  int p1,p2;
  Q();
}
void Q(){
  int q1,q2,q3;
}
```

Once main finishes, its activation record is popped off the stack leaving it empty.

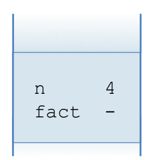


Example 2 - Recursive Function [1]

```
int factorial(int n){
  if(n <= 1)return 1;
  else
    return n*factorial(n-1);
}
```

```
fact(4);
```

The function call `fact(4)` results in the creation of an activation record that is pushed onto the runtime stack.



Example 2 - Recursive Function [2]

```
int factorial(int n){
  if(n <= 1)return 1;
  else
    return n*factorial(n-1);
}
```

`fact(3);`

For the recursive call to `fact(3)` another activation record is pushed onto the runtime stack.

n	4
fact	–
n	3
fact	–

43

Example 2 - Recursive Function [3]

```
int factorial(int n){
  if(n <= 1)return 1;
  else
    return n*factorial(n-1);
}
```

`fact(2);`

For the recursive call to `fact(2)` another activation record is pushed onto the runtime stack.

n	4
fact	–
n	3
fact	–
n	2
fact	–

44

Example 2 - Recursive Function [4]

```
int factorial(int n){
  if(n <= 1)return 1;
  else
    return n*factorial(n-1);
}
```

`fact(1);`

For the recursive call to `fact(1)` another activation record is pushed onto the runtime stack.

n	4
fact	–
n	3
fact	–
n	2
fact	–
n	1
fact	–

45

Example 2 - Recursive Function [5]

```
int factorial(int n){
  if(n <= 1)return 1;
  else
    return n*factorial(n-1);
}
```

`fact(1);`

Once the value for `fact(1)` has been computed, the value is placed into the slot that has been reserved for it.

n	4
fact	–
n	3
fact	–
n	2
fact	–
n	1
fact	1

46

Example 2 - Recursive Function [6]

```
int factorial(int n){
  if(n <= 1)return 1;
  else
    return n*factorial(n-1);
}
```

`fact(2);`

Once the value for `fact(2)` has been computed, the value is placed into the slot that has been reserved for it.

n	4
fact	–
n	3
fact	–
n	2
fact	2

47

Example 2 - Recursive Function [7]

```
int factorial(int n){
  if(n <= 1)return 1;
  else
    return n*factorial(n-1);
}
```

`fact(3);`

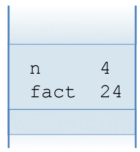
Once the value for `fact(3)` has been computed, the value is placed into the slot that has been reserved for it.

n	4
fact	–
n	3
fact	6

48

Example 2 - Recursive Function [8]

```
int factorial(int n){
  if(n <= 1)return 1;
  else
    return n*factorial(n-1);
}
```



`fact(4);`

Once the value for `fact(3)` has been computed, the value is placed into the slot that has been reserved for it.

49

Queues

- A "regular" queue is an abstract data type which adds elements to an end and removes elements from the other end.
- Queues are FIFO lists (first-in first-out)
- Queues can be implemented using linked lists or arrays.

50

Queue Operations

- **void clear():** remove all the elements
- **<type> dequeue():** remove and return element in front
- **void enqueue(e):** add element e to end of queue
- **bool isEmpty():** true if queue is empty
- **bool isFull():** true if queue is full

51

Longest Common Subsequence

- Given: 2 lists
- Find: Use the Longest Common Subsequence (LCS) to find the difference between them
- Recall that a subsequence preserves order
- Example: L1 = abcabba
  L2 = cbabac
  » LCS = baba or cbba

52

LCS: The `diff` Command

- Find the LCS of lines
- The remaining lines have changed

file1.txt

```
Hello World!
This is file one.
```

file2.txt

```
Hello World!
This is file two.
```

diff file1.txt file2.txt

```
2c2
< This is file one.
---
> This is file two.
```

53

Computing the LCS [1]

- Assume we are comparing prefixes of 2 sequences
  - The prefix of the first sequence is of length i: $a_1,a_2,\ldots a_i$
  - The prefix of the second sequence is of length j: $b_1,b_2,\ldots b_j$
- The empty string is of length 0.

54

Computing the LCS [2]

Recursive definition for 2 prefixes of length i and j:

- Basis: i+j = 0
  Both of the strings must be ε (i=j=0)
  LCS(i,j) = LCS(0,0) = 0
- Induction:
  1) i=0 or j=0 LCS(i, j) = 0
  2) i>0 and j>0 and $a_i \neq b_j$
     LCS(i, j) = max(LCS(i,j-1), LCS(i-1,j))
  3) i>0 and j>0 and $a_i = b_j$
     LCS(i, j) = 1 + LCS(i-1,j-1)

55

Computing the LCS [3]

- Direct implementation from rules would yield an exponential time algorithm.
- It is more efficient to keep track of intermediate results
  - *Dynamic programming* computes small instances first and stores them

56

Longest Common Subsequence

- Given: 2 lists
- Find: Use the Longest Common Subsequence (LCS) to find the difference between them
- **Subsequence:** Subset of the elements of a list preserving the order of their occurrence in the original list (not necessarily contiguous)
- Example: L1 = abcabba
  L2 = cbabac
  » LCS = baba or cbba

57

LCS Example [1]

- Example: x = cbabac and y = abcabba
- Fill matrix row by row

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0							
2	b								
3	a								
4	b								
5	a								
6	c								

- *Initialize row 0 with 0.*
- *Start each row with 0.*

i=0 or j=0

58

LCS Example [2]

i=1 and j=1

x_1=c and y_1=a → $x_1 \neq y_1$

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0						
2	b								
3	a								
4	b								
5	a								
6	c								

2) LCS(i, j) = max(LCS(i,j-1), LCS(i-1,j))

LCS(1,1) = max(LCS(1,0), LCS(0,1)) = max(0, 0) = 0

59

LCS Example [3]

i=1 and j=2

x_1=c and y_2=b → $x_1 \neq y_2$

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0					
2	b								
3	a								
4	b								
5	a								
6	c								

2) LCS(i, j) = max(LCS(i,j-1), LCS(i-1,j))

LCS(1,2) = max(LCS(1,1), LCS(0,2)) = max(0, 0) = 0

60

LCS Example [4]

i=1 and j=3
x_1=c and y_3=c → x_1=y_3

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1				
2	b								
3	a								
4	b								
5	a								
6	c								

3) LCS(i, j) = 1 + LCS(i-1,j-1)

LCS(1,3) = 1 + LCS(0,2) = 1 + 0 = 1

61

LCS Example [5]

i=1 and j=4
x_1=c and y_4=b → $x_1 \neq y_4$

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1			
2	b								
3	a								
4	b								
5	a								
6	c								

2) LCS(i, j) = max(LCS(i,j-1), LCS(i-1,j))

LCS(1,4) = max(LCS(1,3), LCS(0,4)) = max(1, 0) = 1

62

LCS Example [6]

Fill in remaining rows accordingly

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

- *Result: LCS for each pair of prefixes*
- *So how do we recover the actual sequences?*

63

Retrieving the LCS [1]

Recovering the LCS: Start at bottom right corner

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

- *If $x_i = y_j$* **Move to (i-1, j-1)**
- *If $x_i \neq y_j$ and LCS(i-1, j) = LCS(j,j)* **Move to (i-1,j)**
- *If $x_i \neq y_j$ and LCS(i-1, j) ≠ LCS(j,j)* **Move to (i,j-1)**

64

Retrieving the LCS [2]

i=6, j=7 $x_6 \neq y_7$ since c ≠ a
LCS(5,7) = LCS(6,7)

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

- *If $x_i = y_j$* **Move to (i-1, j-1)**
- *If $x_i \neq y_j$ and LCS(i-1, j) = LCS(j,j)* **Move to (i-1,j)**
- *If $x_i \neq y_j$ and LCS(i-1, j) ≠ LCS(j,j)* **Move to (i,j-1)**

65

Retrieving the LCS [3]

i=5, j=7 $x_6 = y_7$ since a = a

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

- *If $x_i = y_j$* **Move to (i-1, j-1)**
- *If $x_i \neq y_j$ and LCS(i-1, j) = LCS(j,j)* **Move to (i-1,j)**
- *If $x_i \neq y_j$ and LCS(i-1, j) ≠ LCS(j,j)* **Move to (i,j-1)**

66

Retrieving the LCS [4]

i=4, j=6 $x_4 = y_6$ since b = b

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

- *If $x_i = y_j$* **Move to (i-1, j-1)**
- *If $x_i \neq y_j$ and LCS(i-1, j) = LCS(j,j)* **Move to (i-1,j)**
- *If $x_i \neq y_j$ and LCS(i-1, j) ≠ LCS(j,j)* **Move to (i,j-1)**

67

Retrieving the LCS [5]

i=3, j=5 $x_3 \neq y_5$ since a ≠ b
LCS(2,5) = LCS(3,5)

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

- *If $x_i = y_j$* **Move to (i-1, j-1)**
- *If $x_i \neq y_j$ and LCS(i-1, j) = LCS(j,j)* **Move to (i-1,j)**
- *If $x_i \neq y_j$ and LCS(i-1, j) ≠ LCS(j,j)* **Move to (i,j-1)**

68

Retrieving the LCS [6]

i=2, j=5 $x_2 = y_5$ since b = b

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

- *If $x_i = y_j$* **Move to (i-1, j-1)**
- *If $x_i \neq y_j$ and LCS(i-1, j) = LCS(j,j)* **Move to (i-1,j)**
- *If $x_i \neq y_j$ and LCS(i-1, j) ≠ LCS(j,j)* **Move to (i,j-1)**

69

Retrieving the LCS [7]

i=1, j=4 $x_1 \neq y_4$ since c ≠ a
LCS(0,4) ≠ LCS(1,4)

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

- *If $x_i = y_j$* **Move to (i-1, j-1)**
- *If $x_i \neq y_j$ and LCS(i-1, j) = LCS(j,j)* **Move to (i-1,j)**
- *If $x_i \neq y_j$ and LCS(i-1, j) ≠ LCS(j,j)* **Move to (i,j-1)**

70

Retrieving the LCS [8]

i=1, j=3 $x_1 = y_3$ since c = c

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

- *If $x_i = y_j$* **Move to (i-1, j-1)**
- *If $x_i \neq y_j$ and LCS(i-1, j) = LCS(j,j)* **Move to (i-1,j)**
- *If $x_i \neq y_j$ and LCS(i-1, j) ≠ LCS(j,j)* **Move to (i,j-1)**

71

Retrieving the LCS [9]

i=0 or j=0 reached

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

72

Retrieving the LCS [10]

Read the sequence

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

- *Select those fields in the path for which $x_i=y_j$:* ***cbba***
- *To obtain alternate result move (i, j-1) when creating the path if $x_i \neq y_j$ and LCS(i, j-1) = LCS(j,j)*

73

Retrieving the LCS – Preference = **Up**

- *When backtracking:*
  - *If letters are equal then go diagonal*
  - *If letters are not equal*
    - *If cell above has same value then go up, otherwise go left*

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

cbba

74

Retrieving the LCS – Preference = **Left**

- *When backtracking:*
  - *If letters are equal then go diagonal*
  - *If letters are not equal*
    - *If cell left has same value then go left, otherwise go up*

		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

baba

75

Summary

- Lists: Linked Lists, Sorted Lists, Doubly Linked Lists, Array-Based Lists
- Stacks
- Queues
- Longest Common Subsequence

76