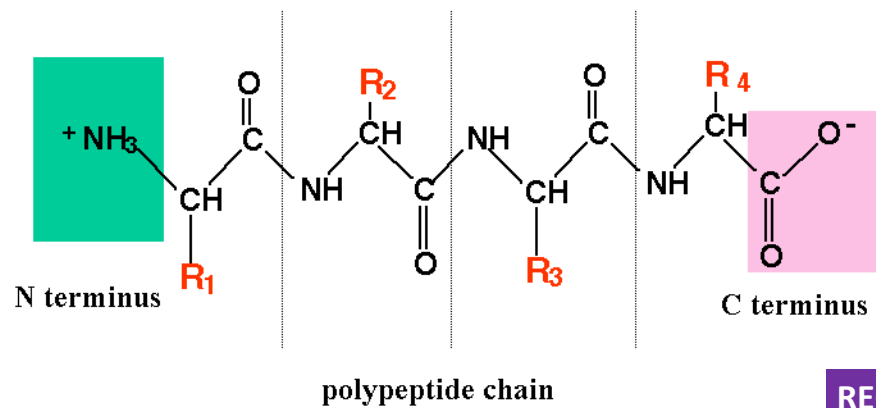


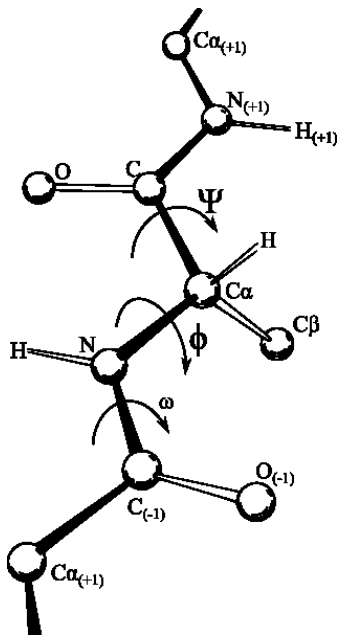
## Lecture 03-04

### Peptide

Peptide = chain of amino acids

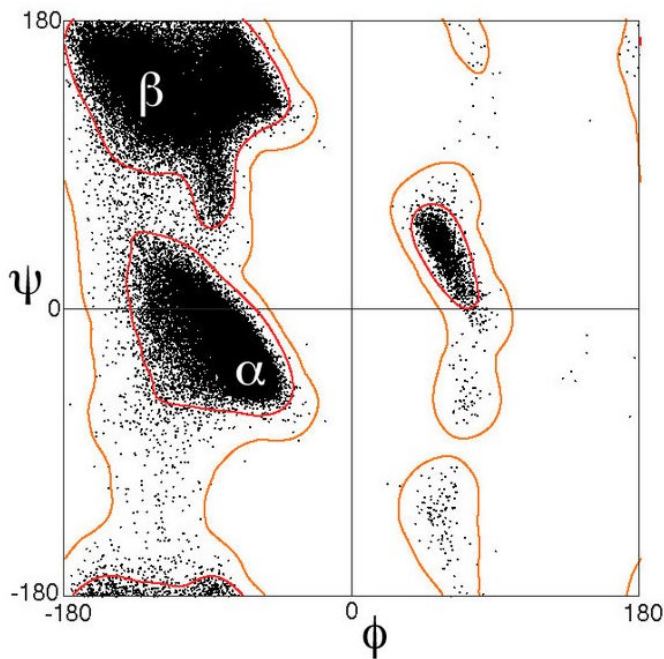


# Protein Backbone



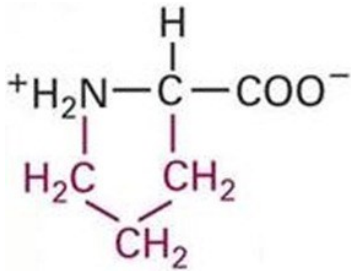
RECAP

# Ramachandran Plot

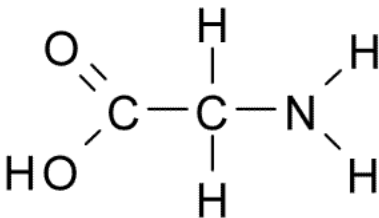


RECAP

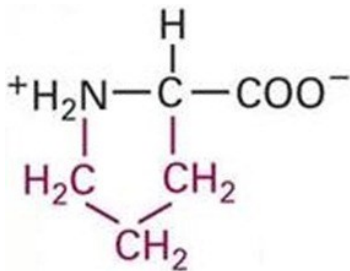
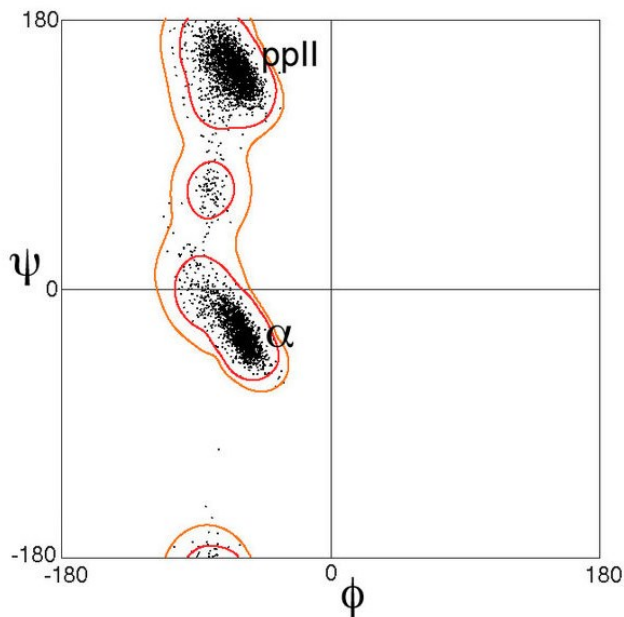
Proline  
structure



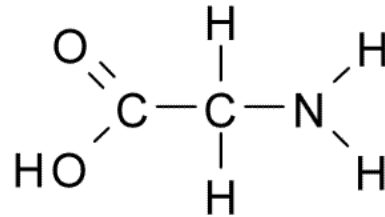
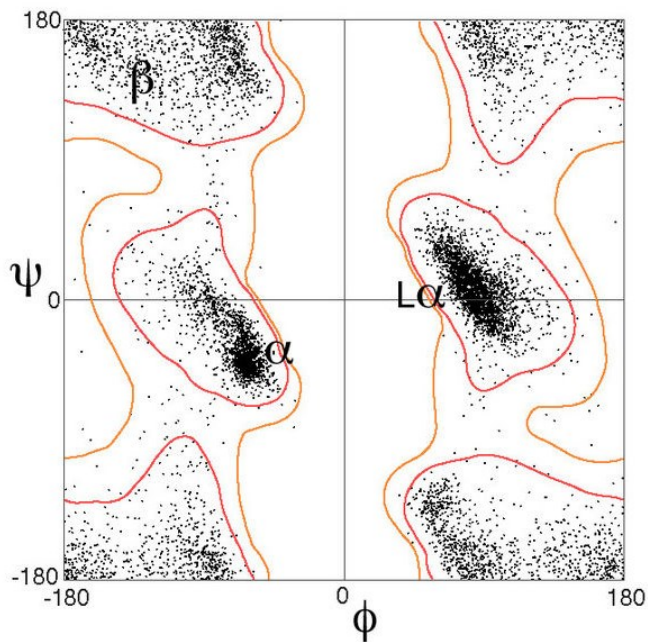
Glycine  
structure



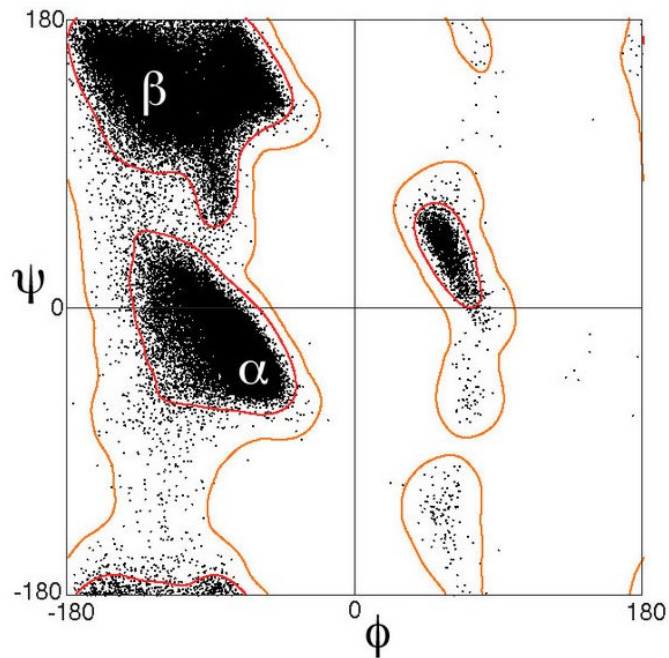
Ramachandran Plot for Proline



## Ramachandran Plot for Glycine



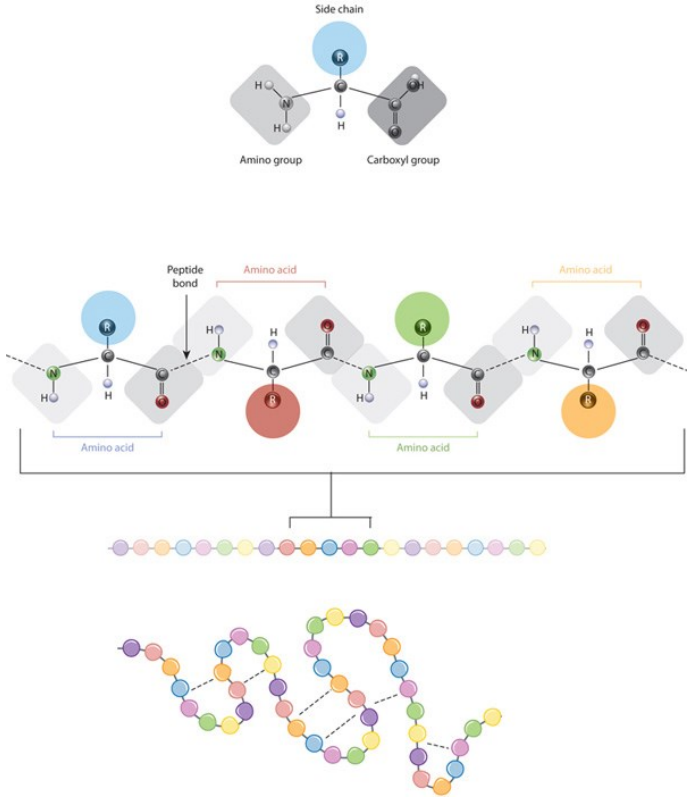
## How to implement Ramachandran Plot?



**Food for your thought**

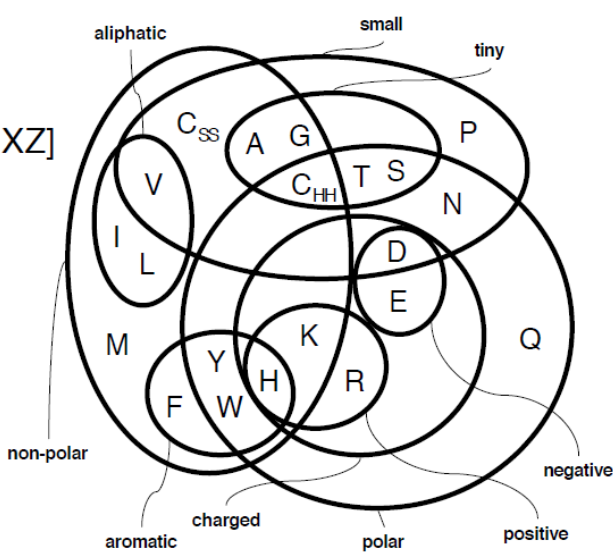
Come up with an efficient data structure to implement for frequent use of Ramachandran Plot.

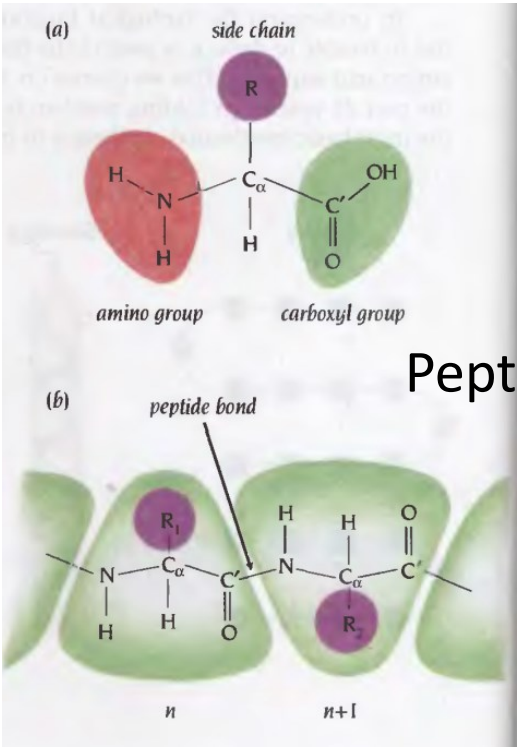
# Peptide



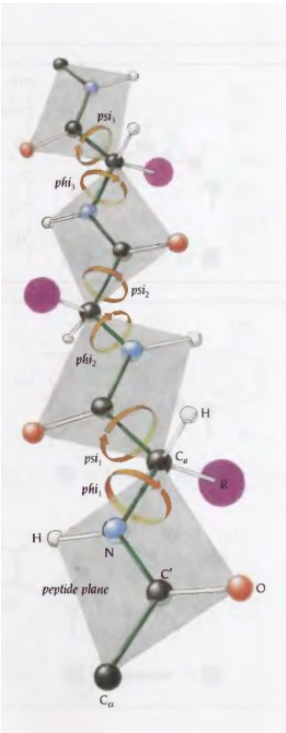
# Proteins

- ❑ 20 amino acids: [A-Z] but not [BJOUXZ]
- ❑ ~300 aa/protein (bacteria), ~200 aa/domain
- ❑ ~200 k known sequences

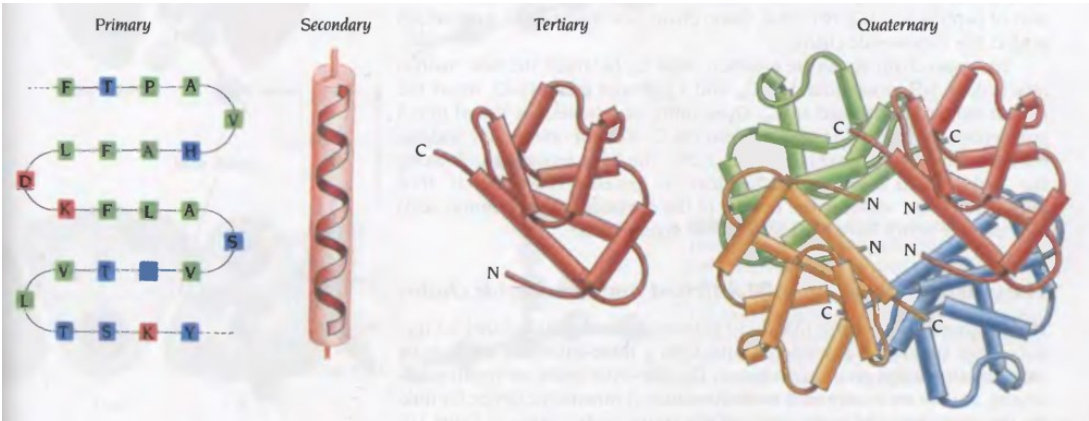




Peptide bond



Proteins

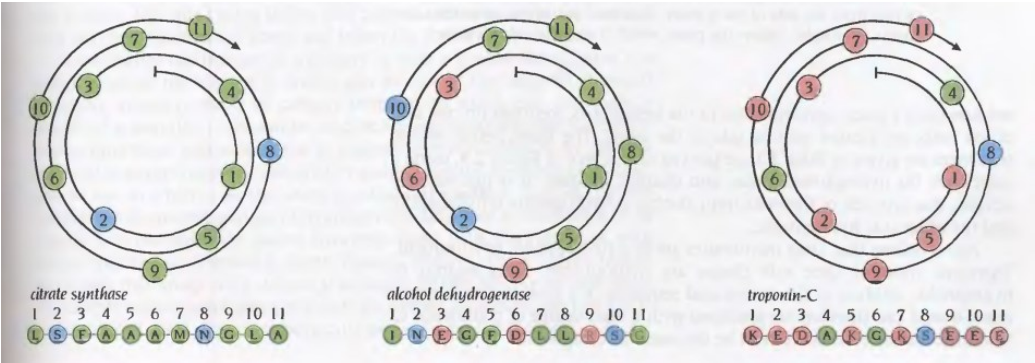




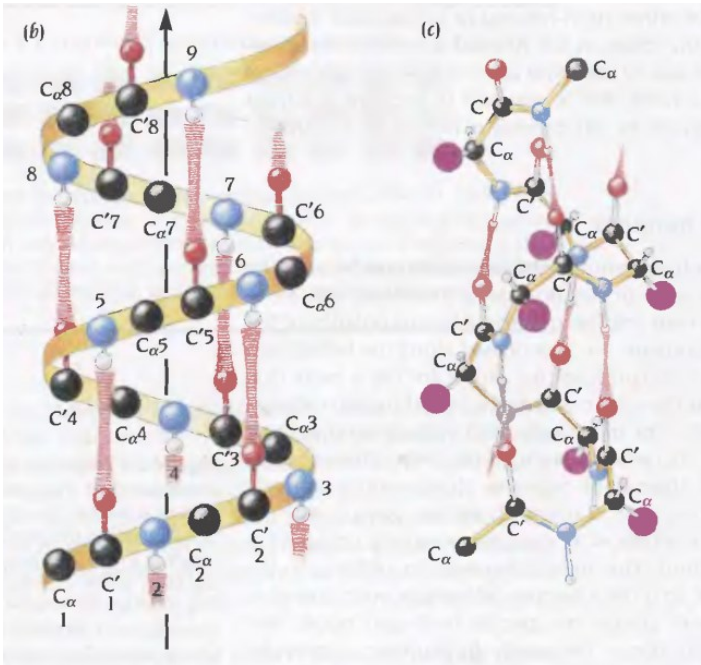
# Preference of AA in helix

**Table 2.1** Amino acid sequences of three  $\alpha$  helices

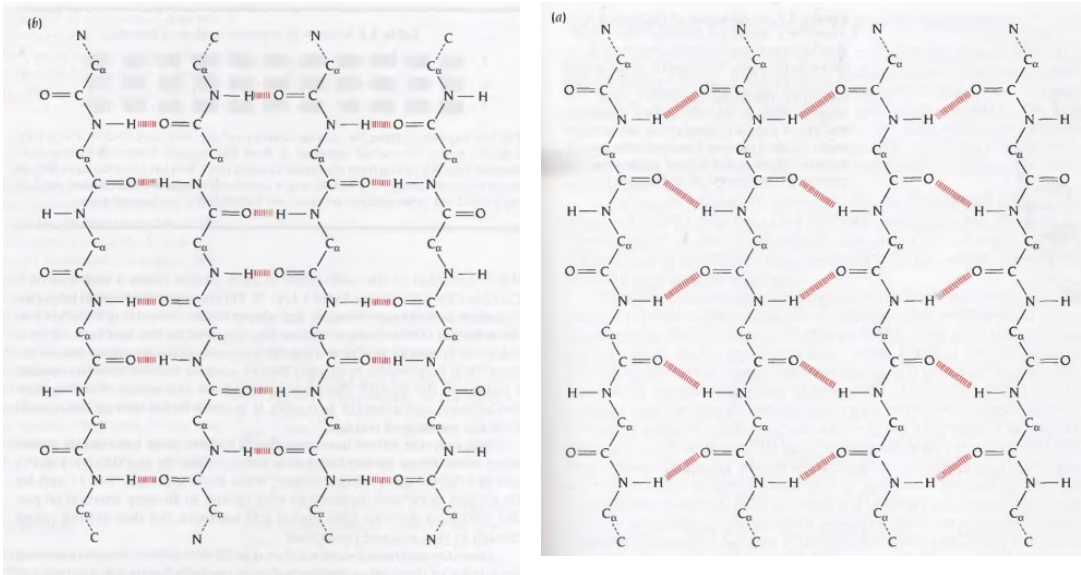
1.	-	Leu	-	Ser	-	Phe	-	Ala	-	Ala	-	Ala	-	Met	-	Asn	-	Gly	-	Leu	-	Ala	-
2.	-	Ile	-	Asn	-	Glu	-	Gly	-	Phe	-	Asp	-	Leu	-	Leu	-	Arg	-	Ser	-	Gly	-
3.	-	Lys	-	Glu	-	Asp	-	Ala	-	Lys	-	Gly	-	Lys	-	Ser	-	Glu	-	Glu	-	Glu	-



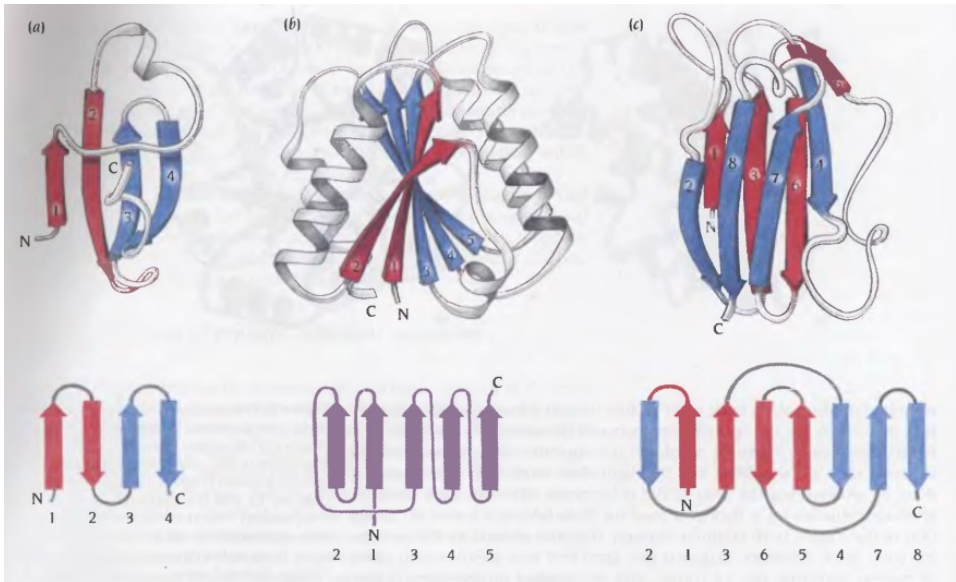
# Formation of helix



# Formation of sheet



# Topology in Structure





## Systems

- **Deterministic System**
  - In a deterministic system no randomness is involved in the development of future states of the system. A deterministic model will thus always produce the same output from a given starting condition or initial state.
- **Dynamical System**
  - A dynamical system is a concept in mathematics where a fixed rule describes the time dependence of a point in a geometrical space. Examples include the mathematical models that describe the swinging of a clock pendulum, the flow of water in a pipe etc.

## Biological Parameter Space

- **N-Dimensional**
  - Temperature
  - Pressure
  - pH
  - Ionic Concentration
  - Solvent
  - Mode of interactions
  - Entropy
  - .....

Value of N varies from problem to problem.

## References

- <http://kinemage.biochem.duke.edu/teaching/anatax/>
- **Introduction to Protein Structure**  
Authors: Carl Branden, John Tooze

## How to Implement a Dictionary?

- Sequences
  - ordered
  - unordered
- Binary Search Trees
- **Hash tables**

## Hashing

- Another important and widely useful technique for implementing dictionaries
- Constant time per operation (on the average)
- Worst case time proportional to the size of the set for each operation (just like array and chain implementation)

## Hashing - Basic Idea

- Use *hash function* to map keys into positions in a *hash table*

### Ideally

- If element  $e$  has key  $k$  and  $h$  is hash function, then  $e$  is stored in position  $h(k)$  of table
- To search for  $e$ , compute  $h(k)$  to locate position. If no element, dictionary does not contain  $e$ .

## Hash function example

- elements = Integers
- $h(i) = i \% 10$
- insert 41, 34, 7, and 18
- constant-time lookup:
  - just look at  $i \% 10$  again later
- Hash tables have no ordering information!
  - Expensive to do following:
    - getMin, getMax, removeMin, removeMax,
    - the various ordered traversals
    - printing items in sorted order

0	
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

## Hashing Operations

- **Search**
  - looks for key  $k$
- **Insert**
  - first searches for a slot, then inserts
- **Delete**
  - Cannot just turn the slot containing the key we want to delete to contain NIL. Why?

## Hashing Analysis

- **Analysis**
  - $O(b)$  time to initialize hash table ( $b$  number of positions or buckets in hash table)
  - $O(1)$  time to perform *insert*, *remove*, *search*
- **Reality**
  - Works for implementing dictionaries, but many applications have key ranges that are too large to have 1-1 mapping between buckets and keys!
  - Example:
    - Suppose key can take on values from 0 .. 65,535 (2 byte unsigned int)
    - Expect  $\approx 1,000$  records at any given time
    - Impractical to use hash table with 65,536 slots!

## Hash Collisions

- **Collision:** the event that two hash table elements map into the same slot in the array

- example: insert 41, 34, 7, 18, then 21
- 21 hashes into the same slot as 41!

- **Resolution:**

- How can we choose the hash function to minimize collisions?
- What do we do about collisions when they occur?

0	
1	21
2	
3	
4	34
5	
6	
7	7
8	18
9	

## Collision Resolution Policies

- Two classes:
  1. Closed hashing / open addressing
  2. Open hashing / separate chaining
- Difference has to do with whether collisions are stored *outside the table* (open hashing) or whether collisions result in storing one of the records at *another slot in the table* (closed hashing)



# Open Addressing

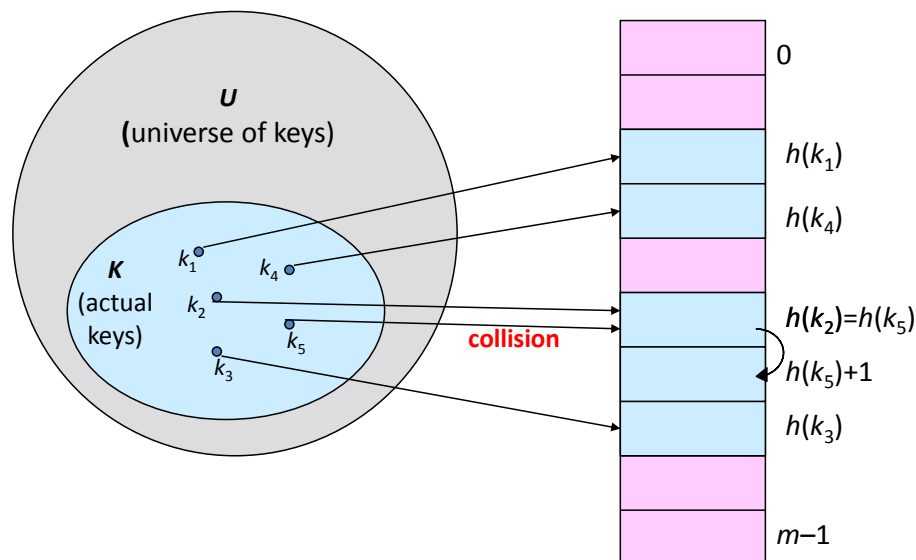
## • Concept:

- Store all  $n$  keys in the  $m$  slots of the hash table itself.
- Each slot contains either a key or NIL.
- To **search** for key  $k$ :
  - Examine slot  $h(k)$ . Examining a slot is known as a **probe**.
  - If slot  $h(k)$  contains key  $k$ , the search is successful. If the slot contains NIL, the search is unsuccessful.
  - There's a third possibility: **slot  $h(k)$  contains a key that is not  $k$ .**
    - Compute the index of some other slot, based on  $k$  and which probe we are on.
    - Keep probing until we either find key  $k$  or we find a slot holding NIL.

**Advantages:** Avoids pointers; so less code, and we can dedicate the memory to the table.

What can you say about the load factor  $\alpha = n/m$ ?

## Open addressing - issue



## Closed Hashing

- Associated with closed hashing is a *rehash strategy*:  
“If we try to place  $x$  in bucket  $h(x)$  and find it occupied, find alternative location  $h_1(x)$ ,  $h_2(x)$ , etc. Try each in order, if none empty table is full,”
- $h(x)$  is called *home bucket*
- Simplest rehash strategy is called *linear hashing*  
$$h_i(x) = (h(x) + i) \% D$$
- In general, our collision resolution strategy is to generate a sequence of hash table slots (probe sequence) that can hold the record; test each slot until find empty one (probing)

## Importance of Good Hash Functions

- Recall the assumption of *simple uniform hashing*:
  - Any key is equally likely to hash into any of the slots, independent of where any other key hashes to.
  - $O(1)$  time to compute  $h(k)$ .
- Hash values should be independent of any patterns that might exist in the data.
  - E.g. If each key is drawn independently from  $U$  according to a probability distribution  $P$ , we want  
for all  $j \in [0 \dots m-1]$ ,  $\sum_{k: h(k)=j} P(k) = 1/m$
- Often use heuristics, based on the domain of the keys, to create a hash function that performs well.

# Two examples only

- Division method**
  - Map each key  $k$  into one of the  $m$  slots by taking the remainder of  $k$  divided by  $m$ .
$$h(k) = k \bmod m$$
  - Example:  $m = 31$  and  $k = 78 \Rightarrow h(k) = 16$ .
  - Advantage: Fast, since requires just one division operation.
  - Disadvantage: For some values, such as  $m=2^p$ , the hash depends on just a subset of the bits of the key.
  - Note: Primes are good, if not too close to power of 2 (or 10).
- Multiplication method**
  - Map each key  $k$  to one of the  $m$  slots indicated by the fractional part of  $k$  times a chosen real  $0 < A < 1$ .
$$h(k) = \lfloor m (kA \bmod 1) \rfloor = \lfloor m (kA - \lfloor kA \rfloor) \rfloor$$
  - Example:  $m = 1000, k = 123, A \approx 0.6180339887...$ 
$$h(k) = \lfloor 1000(123 \cdot 0.6180339887 \bmod 1) \rfloor$$
$$= \lfloor 1000 \cdot 0.0181... \rfloor = 18.$$
  - Disadvantage: A bit slower than the division method.
  - Advantage: Value of  $m$  is not critical.

**Homework**

Implement these two techniques.

## Example Linear (Closed) Hashing

- $D=8$ , keys  $a,b,c,d$  have hash values  $h(a)=3, h(b)=0, h(c)=4, h(d)=3$
- Where do we insert  $d$ ? 3 already filled
- Probe sequence using linear hashing:
$$h_1(d) = (h(d)+1)\%8 = 4\%8 = 4$$
$$h_2(d) = (h(d)+2)\%8 = 5\%8 = 5^*$$
$$h_3(d) = (h(d)+3)\%8 = 6\%8 = 6$$
etc.  
7, 0, 1, 2
- Wraps around the beginning of the table!

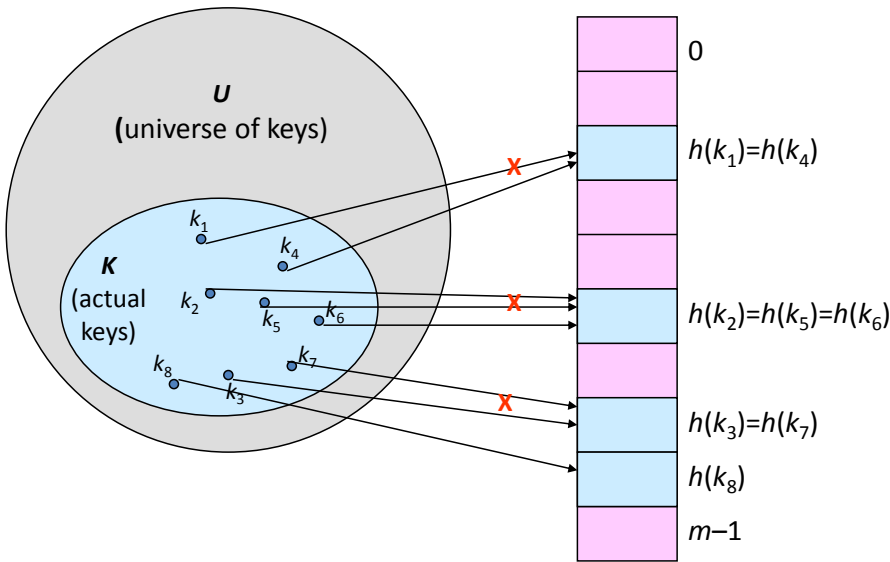
0	b
1	
2	
3	a
4	c
5	d
6	
7	

**Performance Analysis**

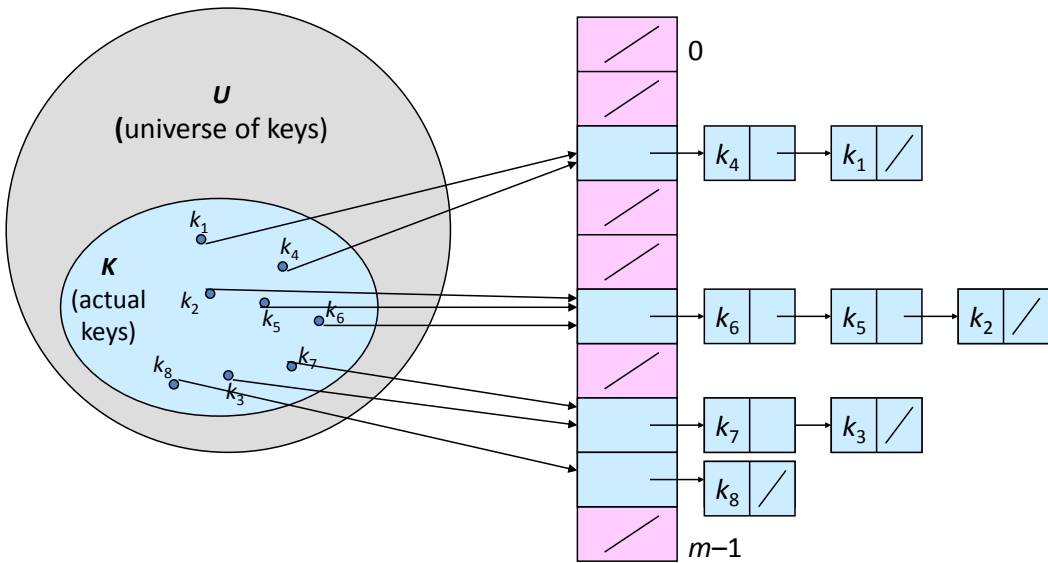
Computational complexity for initialization?

Computational complexity for insertion / search?

## Collision Resolution by Chaining



## Collision Resolution by Chaining



## Hashing with Chaining

### Dictionary Operations:

- Chained-Hash-Insert ( $\mathcal{T}, x$ )
  - Insert  $x$  at the head of list  $\mathcal{T}[h(\text{key}[x])]$ .
  - Worst-case complexity:  $O(1)$ .
- Chained-Hash-Search ( $\mathcal{T}, k$ )
  - Search an element with key  $k$  in list  $\mathcal{T}[h(k)]$ .
  - Worst-case complexity: proportional to length of list.
- Chained-Hash-Delete ( $\mathcal{T}, x$ )
  - Delete  $x$  from the list  $\mathcal{T}[h(\text{key}[x])]$ .
  - Worst-case complexity: search time +  $O(1)$ .
    - Need pointer to preceding element, or a doubly-linked list.

## Analysis of Chained-Hash-Search

- ✓ **Worst-case search time:** time to compute  $h(k) + \Theta(n)$ .
- ✓ **Average time:** depends on how  $h$  distributes keys among slots.
  - ✓ **Assumptions:**
    - *Simple uniform hashing:* Any key is equally likely to hash into any of the slots, independent of where any other key hashes to.
    - $O(1)$  time to compute  $h(k)$ .
  - ✓ **Define** Load factor  $\alpha = n/m$  = average # of keys per slot.
    - $n$  – number of keys stored in the hash table.
    - $m$  – number of slots = # linked lists.

## Implications for separate chaining

- If  $n = O(m)$ , then load factor  $\alpha = n/m = O(m)/m = O(1)$ .
- Deletion takes  $O(1)$  worst-case time if you have a pointer to the preceding element in the list.
- Hence, for hash tables with chaining, all dictionary operations take  $O(1)$  time on average, given the assumptions of simple uniform hashing and  $O(1)$  time hash function evaluation.
- Extra memory needed for linked list pointers.
- Can we satisfy the simple uniform hashing assumption?

## Probe Sequence

- Sequence of slots examined during a key search constitutes a *probe sequence*.
- Probe sequence must be a permutation of the slot numbers.
  - We examine every slot in the table, if we have to.
  - We don't examine any slot more than once.
- One way to think of it: extend hash function to:
  - $h : U \times \underbrace{\{0, 1, \dots, m-1\}}_{\text{probe number}} \rightarrow \underbrace{\{0, 1, \dots, m-1\}}_{\text{slot number}}$

Universe of Keys



## Computing Probe Sequences

- The ideal situation is *uniform hashing*:
  - Generalization of simple uniform hashing.
  - Each key is equally likely to have any of the  $m!$  permutations of  $\langle 0, 1, \dots, m-1 \rangle$  as its probe sequence.
  - It is hard to implement true uniform hashing.
- Approximate with techniques that guarantee to probe a permutation of  $[0 \dots m-1]$ , even if they don't produce all  $m!$  probe sequences
  - Linear Probing.
  - Quadratic Probing.
  - Double Hashing.

## Lecture 05

## Hash function example

- elements = Integers
- $h(i) = i \% 10$
- insert 41, 34, 7, and 18
- constant-time lookup:
  - just look at  $i \% 10$  again later
- Hash tables have no ordering information!
  - Expensive to do following:
    - getMin, getMax, removeMin, removeMax,
    - the various ordered traversals
    - printing items in sorted order

0	
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

RECAP

## Hashing Operations

- **Search**
  - looks for key  $k$
- **Insert**
  - first searches for a slot, then inserts
- **Delete**
  - Cannot just turn the slot containing the key we want to delete to contain NIL. Why?

RECAP

## Hash Collisions

- **Collision:** the event that two hash table elements map into the same slot in the array
  - example: insert 41, 34, 7, 18, then 21
  - 21 hashes into the same slot as 41!
- **Resolution:**
  - How can we choose the hash function to minimize collisions?
  - What do we do about collisions when they occur?

0	
1	21
2	
3	
4	34
5	
6	
7	7
8	18
9	

RECAP

## Collision Resolution Policies

- Two classes:
  1. Closed hashing / open addressing
  2. Open hashing / separate chaining
- Difference has to do with whether collisions are stored *outside the table* (open hashing) or whether collisions result in storing one of the records at *another slot in the table* (closed hashing)

RECAP

# Open Addressing

- **Concept:**
  - Store all  $n$  keys in the  $m$  slots of the hash table itself.
  - Each slot contains either a key or NIL.
  - To **search** for key  $k$ :
    - Examine slot  $h(k)$ . Examining a slot is known as a **probe**.
    - If slot  $h(k)$  contains key  $k$ , the search is successful. If the slot contains NIL, the search is unsuccessful.
    - There's a third possibility: **slot  $h(k)$  contains a key that is not  $k$** .
      - Compute the index of some other slot, based on  $k$  and which probe we are on.
      - Keep probing until we either find key  $k$  or we find a slot holding NIL.

RECAP

Advantages: Avoids pointers; so less code, and we can dedicate the memory to the table.

What can you say about the load factor  $\alpha = n/m$ ?

## Example Linear (Closed) Hashing

- $D=8$ , keys  $a,b,c,d$  have hash values  $h(a)=3$ ,  $h(b)=0$ ,  $h(c)=4$ ,  $h(d)=3$
- Where do we insert  $d$ ? 3 already filled
- Probe sequence using linear hashing:
  - $h_1(d) = (h(d)+1)\%8 = 4\%8 = 4$
  - $h_2(d) = (h(d)+2)\%8 = 5\%8 = 5^*$
  - $h_3(d) = (h(d)+3)\%8 = 6\%8 = 6$
  - etc.
- 7, 0, 1, 2
- Wraps around the beginning of the table!

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

Performance Analysis

Computational complexity for initialization?  
Computational complexity for insertion / search?

RECAP

## Hashing with Chaining

### Dictionary Operations:

- Chained-Hash-Insert ( $T, x$ )
  - Insert  $x$  at the head of list  $T[h(key[x])]$ .
  - Worst-case complexity:  $O(1)$ .
- Chained-Hash-Search ( $T, k$ )
  - Search an element with key  $k$  in list  $T[h(k)]$ .
  - Worst-case complexity: proportional to length of list.
- Chained-Hash-Delete ( $T, x$ )
  - Delete  $x$  from the list  $T[h(key[x])]$ .
  - Worst-case complexity: search time +  $O(1)$ .
    - Need pointer to preceding element, or a doubly-linked list.

RECAP

## Linear Probing

- $h(k, i) = (h(k, 0) + i) \bmod m$ 

$\nearrow$   
key

$\nearrow$   
Probe number

$\nearrow$   
Original hash function
- The initial probe determines the entire probe sequence.
- Suffers from *primary clustering*:
  - Long runs of occupied sequences build up.
  - Long runs tend to get longer, since an empty slot preceded by  $i$  full slots gets filled next with probability  $(i+1)/m$ .

## Clustering problem

- **Clustering:** nodes being placed close together by probing, which degrades hash table's performance
  - add 89, 18, 49, 58, 9
  - now searching for the value 28 will have to check half the hash table! no longer constant time...

0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

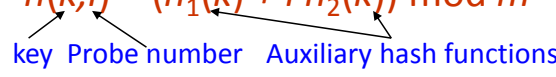
## Quadratic Probing

- $h(k,i) = (h'(k) + c_1i + c_2i^2) \bmod m$   $c_1 \neq c_2$
- Can suffer from *secondary clustering*
- **Example:** resolving collisions on slot  $i$  by putting the colliding element into slot  $i+1$ ,  $i+4$ ,  $i+9$ ,  $i+16$ , ...
  - add 89, 18, 49, 58, 9
    - 49 collides (89 is already there), so we search ahead by +1 to empty slot 0
    - 58 collides (18 is already there), so we search ahead by +1 to occupied slot 9, then +4 to empty slot 2
    - 9 collides (89 is already there), so we search ahead by +1 to occupied slot 0, then +4 to empty slot 3
  - clustering is reduced
  - what is the lookup algorithm?

0	49
1	
2	58
3	9
4	
5	
6	
7	
8	18
9	89



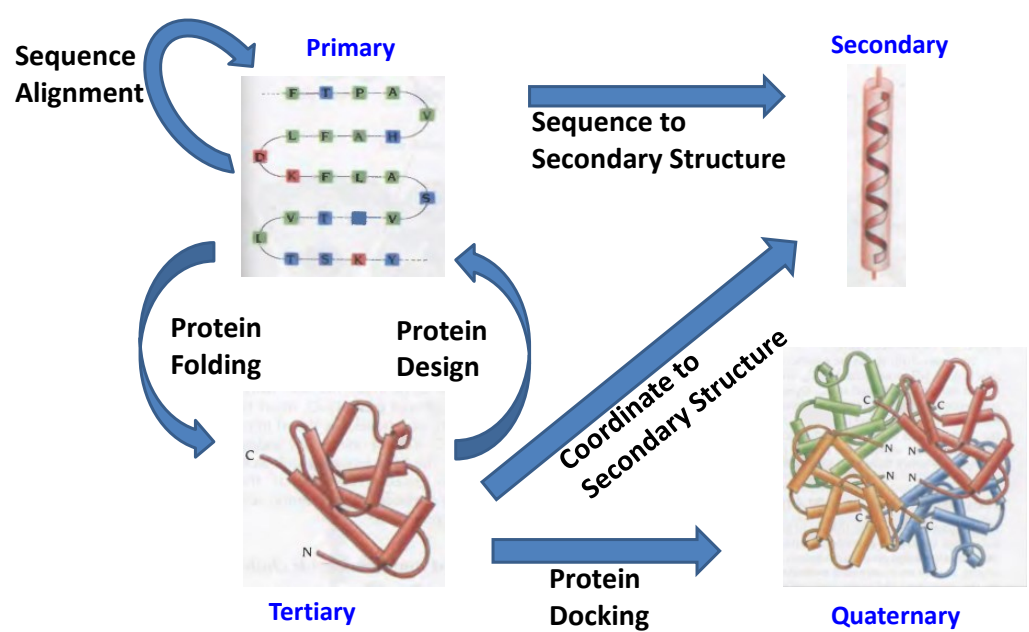
## Double Hashing

- $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$   

- Two auxiliary hash functions.
  - $h_1$  gives the initial probe.  $h_2$  gives the remaining probes.
- Must have  $h_2(k)$  relatively prime to  $m$ , so that the probe sequence is a full permutation of  $\langle 0, 1, \dots, m-1 \rangle$ .
  - Choose  $m$  to be a power of 2 and have  $h_2(k)$  always return an odd number. Or,
  - Let  $m$  be prime, and have  $1 < h_2(k) < m$ .
- $\Theta(m^2)$  different probe sequences.
  - One for each possible combination of  $h_1(k)$  and  $h_2(k)$ .
  - Close to the ideal uniform hashing.

## Analysis of hash tables

- Main operation: lookup of item in table
- What is worst-case cost of finding an item?
- Is the worst-case cost different for chaining, and the various open addressing schemes?
- Worst-case analysis doesn't make sense for hash tables, look at average case cost
- Cost highly depend on the **load factor** (no. of elements / array size)
- Which is better – hashing or tree based representation?

# Computational Methods in Proteins



## Aligning / Matching



## Matching / Alignment

- **Longest common substring problem**

Let  $\Sigma$  be an alphabet (finite set; for *DNA alphabet* ( $\Sigma = \{A, C, G, T\}$ )). Search a pattern from the text where both the pattern and text are arrays of elements of  $\Sigma$ .

— *Example:*

- Pattern: *ins, india, iit, iit kharagpur*
- String: *indian institute of technology kharagpur*

- **Longest common subsequence problem**

Let  $\Sigma$  be an alphabet (finite set; for *DNA alphabet* ( $\Sigma = \{A, C, G, T\}$ )). Find the longest subsequence common to all sequences in a set of sequences (often just two sequences) constructed over alphabet set  $\Sigma$ .

— *Example:*

- Pattern: *ins, india, iit, iit kharagpur*
- String: *indian institute of technology kharagpur*

## *Longest common substring problem*

Algorithm	Preprocessing Time	Searching Time	Space Required
Naïve string search algorithm	None	$O(mn)$	None
Rabin–Karp string search algorithm	$O(m)$	$O(n+m)$ $O((n-m)m)$	Constant
Knuth–Morris–Pratt algorithm	$O(m)$	$O(n)$	$O(m)$

Length of a string:  $n$

Length of the pattern:  $m$

Cardinality of character set:  $k$

## *Longest common substring problem*

- Naïve string search algorithm

for a character in **txt**

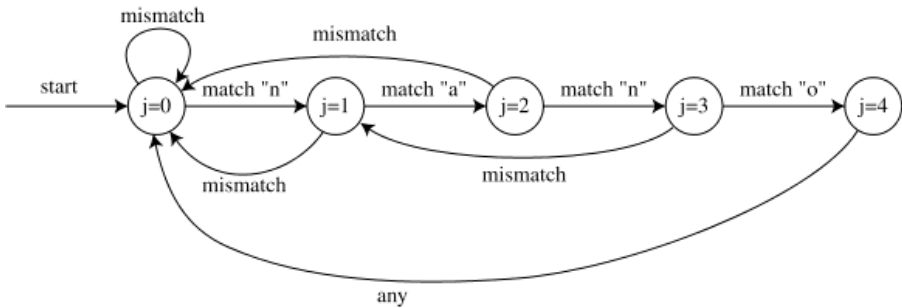
for every character in **pat**

break the search if **txt** character does not **match** **pat** character.

If all the character of **pat** **matches** then output a match.

Longest common substring problem

- Knuth–Morris–Pratt (KMP) algorithm



Pattern: *nano*  
Text: *banananobano*

Workout  
Implement

A Rabin-Karp example

- Given  $T = 31415926535$  and  $P = 26$
- We choose  $q = 11$
- $P \bmod q = 26 \bmod 11 = 4$

Complexity?

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$31 \bmod 11 = 9$  not equal to 4

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$14 \bmod 11 = 3$  not equal to 4

3	1	4	1	5	9	2	6	5	3	5
---	---	---	---	---	---	---	---	---	---	---

$41 \bmod 11 = 8$  not equal to 4

## *Applications*

- Alpha hemoglobin and beta hemoglobin are subunits that make up a protein called hemoglobin in red blood cells. Notice the similarities between the two sequences, which probably signify functional similarity.
- Many distantly related proteins have domains that are similar to each other, such as the DNA binding domain or cation binding domain. To find regions of high similarity within multiple sequences of proteins, local alignment must be performed. The local alignment of sequences may provide information of similar functional domains present among distantly related proteins.